

7 Dynamic programming

In this chapter, we will present two dynamic-programming algorithms for scheduling arbitrary precedence graphs with non-uniform deadlines subject to unit-length communication delays. Using these algorithms, we can construct minimum-tardiness schedules for arbitrary precedence graphs. In Section 7.1, an algorithm of Fulkerson [29] is presented that decomposes precedence graphs of width w into w disjoint chains. Such chain decompositions are used by the dynamic-programming algorithms that are presented in Sections 7.2 and 7.4. The first algorithm is presented in Section 7.2. This dynamic-programming algorithm constructs minimum-tardiness schedules for instances (G, m, D_0) . It is similar to the dynamic-programming algorithm presented by Möhring [67] that constructs minimum-length communication-free schedule for precedence graphs with unit-length tasks and the dynamic-programming algorithm of Veltman [87] that constructs minimum-length schedules for precedence graphs with unit-length tasks subject to unit-length communication delays. Like the algorithms of Möhring [67] and Veltman [87], the time complexity of the algorithm presented in Section 7.2 is exponential in the width of the precedence graph. Hence it constructs minimum-tardiness schedules in polynomial time for precedence graphs of bounded width.

Sections 7.3 and 7.4 are concerned with scheduling precedence graphs with arbitrary task lengths. In Section 7.3, it is proved that constructing a minimum-tardiness schedule for a precedence graph of width w on less than w processors is an NP-hard optimisation problem. In Section 7.4, a second dynamic-programming algorithm is presented. This algorithm constructs minimum-tardiness schedules for precedence graphs of width w on at least w processors. Like the algorithm presented in Section 7.2, the time complexity of this algorithm is exponential in the width of the precedence graph, but it constructs minimum-tardiness schedules for precedence graphs of bounded width.

7.1 Decompositions into chains

In this section, we will show how a precedence graph can be decomposed into disjoint chains. Every precedence graph can be viewed as a collection of disjoint chains with precedence constraints between tasks in different chains: every precedence graph with n tasks can be considered as the disjoint union of n chains consisting of one task. Obviously, precedence graphs that do not consist of n pairwise incomparable tasks can be decomposed into a smaller number of chains.

Definition 7.1.1. Let G be a precedence graph. A *chain decomposition* of G is a collection of disjoint chains C_1, \dots, C_k in G , such that $C_1 \cup \dots \cup C_k = V(G)$.

Let C_1, \dots, C_k be a chain decomposition of a precedence graph G . Then C_1, \dots, C_k will be called a chain decomposition of G into k chains.

Example 7.1.2. Let G be the precedence graph shown in Figure 7.1. It is easy to see that G is a precedence graph of width two. Figure 7.1 shows a chain decomposition of G into two disjoint chains $C_1 = \{c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}, c_{1,5}, c_{1,6}\}$ and $C_2 = \{c_{2,1}, c_{2,2}, c_{2,3}, c_{2,4}\}$. A chain decomposition of G into two disjoint chains is not unique: other chain decompositions of G consisting of two

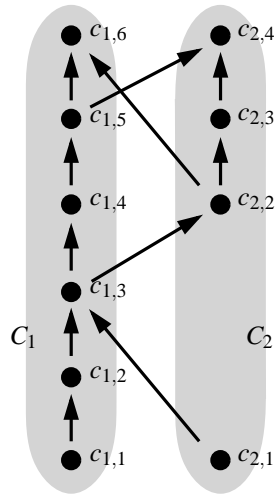


Figure 7.1. A chain decomposition of a precedence graph of width two into two chains

chains are formed by the chains $C'_1 = \{c_{1,1}, c_{1,2}, c_{2,2}, c_{2,3}, c_{2,4}\}$ and $C'_2 = \{c_{2,1}, c_{1,3}, c_{1,4}, c_{1,5}, c_{1,6}\}$ and by the chains $C''_1 = \{c_{1,1}, c_{1,2}, c_{1,3}, c_{2,2}, c_{2,3}, c_{2,4}\}$ and $C''_2 = \{c_{2,1}, c_{1,4}, c_{1,5}, c_{1,6}\}$.

Because a precedence graph of width w contains w pairwise incomparable tasks and incomparable tasks cannot be elements of one chain, a precedence graph of width w cannot be decomposed into less than w chains. Dilworth [22] proved that a precedence graph of width w can be viewed as the disjoint union of exactly w chains.

Theorem 7.1.3. *Let G be a precedence graph of width w . There is a chain decomposition of G into w disjoint chains.*

A chain decomposition of a precedence graph of width w into w chains will be used by the dynamic-programming algorithms presented in Sections 7.2 and 7.4. Dilworth's proof [22] of Theorem 7.1.3 is not constructive, but the proof by Fulkerson [29] is. In his proof of Dilworth's decomposition theorem, Fulkerson presented Algorithm CHAIN DECOMPOSITION shown in Figure 7.2 and proved that it constructs chain decompositions of precedence graphs of width w into w chains.

Algorithm CHAIN DECOMPOSITION works as follows. For a precedence graph G , it constructs an undirected bipartite graph H that contains an edge for every pair of comparable tasks of G and computes a maximum matching of H . This matching is used to construct a chain decomposition of G into disjoint chains.

The time complexity of Algorithm CHAIN DECOMPOSITION can be determined as follows. Let G be a precedence graph of width w . To obtain a better time complexity, we will distinguish two cases depending on whether G is known to be a transitive closure or not. If it is unknown whether G is a transitive closure, then Algorithm CHAIN DECOMPOSITION should start by com-

Algorithm CHAIN DECOMPOSITION

Input. A precedence graph G of width w , such that $V(G) = \{u_1, \dots, u_n\}$.

Output. A chain decomposition C_1, \dots, C_w of G .

1. $V := \{a_1, \dots, a_n\} \cup \{b_1, \dots, b_n\}$
2. $E := \{(a_i, b_j) \mid u_i \prec_G u_j\}$
3. let H be the undirected bipartite graph (V, E)
4. let M be a maximum matching of H
5. $E' := \{(u_i, u_j) \mid (a_i, b_j) \in M\}$
6. let G' be the precedence graph $(V(G), E')$
7. $i := 1$
8. **while** G' contains unmarked tasks
9. **do** let u be an unmarked source of G'
10. $C_i := \{v \in V(G) \mid \text{there is a path from } u \text{ to } v \text{ in } G'\}$
11. mark all tasks in C_i
12. $i := i + 1$

Figure 7.2. Algorithm CHAIN DECOMPOSITION

putting the transitive closure of G . This takes $O(n + e + ne^-)$ time [37]. In the transitive reduction of a precedence graph of width w , every task has at most w children. Hence $e^- \leq wn$. So the transitive closure of G can be constructed in $O(wn^2)$ time. In the remainder of the analysis of the time complexity of Algorithm CHAIN DECOMPOSITION, we will assume that G is a transitive closure.

Since G is a transitive closure, the bipartite graph H can be constructed in $O(n + e^+)$ time. Since $e^+ \leq n^2$, H is constructed in $O(wn^2)$ time. Hopcroft and Karp [48] presented an algorithm that computes a maximum matching in $O(e\sqrt{n})$ time for bipartite graphs with n nodes and e edges. Alt et al. [5] presented an algorithm whose time complexity is better for dense graphs: their algorithm constructs a maximum matching of a bipartite graph in $O(n\sqrt{ne/\log n})$ time.

The number of edges of H equals e^+ . As a result, a maximum matching M of H can be constructed in $O(\min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time. Because the maximum matching of H contains at most n edges, constructing the precedence graph G' takes $O(n)$ time. G' is a chain-like task system. Since every task in G' has indegree and outdegree at most one, constructing the chains in G from G' takes $O(n)$ time. So constructing a chain decomposition of G into w disjoint chains takes $O(wn^2 + \min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time.

Lemma 7.1.4. *For all precedence graphs G of width w , Algorithm CHAIN DECOMPOSITION constructs a chain decomposition of G into w chains in $O(wn^2 + \min\{e^+\sqrt{n}, n\sqrt{ne^+/\log n}\})$ time.*

Let G be a precedence graph of width w . Since G can be decomposed into w disjoint chains, G contains a chain that contains at least $\frac{n}{w}$ tasks. The transitive closure of a chain containing at least $\frac{n}{w}$ tasks contains at least $\frac{n(n-w)}{2w^2}$ arcs. So G^+ contains at least $\frac{n(n-w)}{2w^2}$ arcs. If w is a constant, then G^+ contains $\Theta(n^2)$ arcs. Hence using the algorithm of Alt et al. [5], a chain decomposition of a precedence graph of bounded width can be constructed in $O(n^2\sqrt{n/\log n})$ time.

Lemma 7.1.5. *For all precedence graphs G of constant width w , Algorithm CHAIN DECOMPOSITION constructs a chain decomposition of G into w chains in $O(n^2 \sqrt{n/\log n})$ time.*

7.2 A dynamic-programming algorithm

In this section, a dynamic-programming algorithm will be presented that constructs minimum-tardiness schedules for instances (G, m, D_0) . For precedence graphs of width w , it constructs a minimum-tardiness schedule in $O(n^{w+3})$ time. Hence minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time. The same approach can be used to construct schedules that are optimal with respect to other objective functions (including the minimisation of the makespan) without increasing the time complexity [91]. This leads to an improvement over a result presented by Veltman [87], who showed that minimum-length schedules for precedence graphs of width w can be constructed in $O(n^{2w})$ time.

The time complexity of the dynamic-programming algorithm is exponential in the width of the precedence graph. It is unlikely that there is an algorithm that constructs minimum-length schedules in $O(n^c)$ time, where c is a constant independent of the width of the precedence graph: Bodlaender and Fellows [9] proved that constructing a minimum-length communication-free schedule for arbitrary precedence graphs on k processors is $W[2]$ -hard, where $W[2]$ is the second class of the W -hierarchy for parametrised problems introduced by Downey and Fellows [23]. This implies that it is unlikely that for all fixed positive integers k , a minimum-length schedule for a precedence graph on k processors can be constructed in $O(n^c)$ time for some constant c . In fact, Bodlaender and Fellows [9] proved that constructing minimum-length communication-free schedules for precedence graphs of width $k + 1$ on k processors is $W[2]$ -hard. Their result can be easily generalised for scheduling subject to unit-length communication delays with the objective of minimising the maximum tardiness.

Dynamic programming is a method of constructing an optimal solution of a problem by extending or combining optimal solutions of subproblems. In dynamic programming, the optimal solutions of the subproblems are stored in a table that has an entry for every (relevant) subproblem. The table is then used to construct the best extension or combination of the optimal solutions of the subproblems.

A feasible schedule S for an instance (G, m, D_0) is a list of time slots $(S_0, \dots, S_{\ell-1})$. For each time t , $\bigcup_{i=0}^{t-1} S_i$ is a prefix of G and (S_0, \dots, S_{t-1}) is a feasible schedule for $(G[\bigcup_{i=0}^{t-1} S_i], m, D_0)$. (S_0, \dots, S_{t-1}) will be called a *partial schedule* for (G, m, D_0) . Any schedule S_U for $(G[U], m, D_0)$, such that U is a prefix of G , can be extended to a feasible schedule for (G, m, D_0) by scheduling the remaining tasks after the completion time of the last task of U . So a (minimum-tardiness) schedule for (G, m, D_0) can be constructed by starting with an empty schedule and repeatedly adding the next time slot.

This is the basis of the dynamic-programming algorithm presented in this section: a table containing information about the structure and tardiness of minimum-tardiness partial schedules of (G, m, D_0) is constructed and used to construct a minimum-tardiness schedule for (G, m, D_0) .

Let $S = (S_0, \dots, S_{\ell-1})$ be a minimum-tardiness schedule for (G, m, D_0) . Then for all times

$t \in \{0, \dots, \ell - 1\}$, (S_0, \dots, S_{t-1}) is a feasible schedule for $(G[\bigcup_{i=0}^{t-1} S_i], m, D_0)$ and S_t is a set of sources of $G[V(G) \setminus \bigcup_{i=0}^{t-1} S_i]$. So for each task u in S_t , at most one parent of u is an element of S_{t-1} and for each task u in S_{t-1} , at most one child of u is an element of S_t .

The basic idea of extending partial schedules is the following. Let U be a prefix of G and let (S_0, \dots, S_{t-1}) be a feasible schedule for $(G[U], m, D_0)$. Then a set of sources V of $G[V(G) \setminus U]$ is called *available* with respect to S if

1. $|V| \leq m$;
2. for all tasks u in V , at most one parent of u finishes at time t ; and
3. for all tasks u in U , if u finishes at time t , then V contains at most one child of u .

Note that the availability of V only depends on the size of V and the tasks in U that finish at time t . Hence V will also be called available with respect to (U, S_{t-1}) .

If V is available with respect to (U, S_{t-1}) , then the schedule (S_0, \dots, S_{t-1}, V) is a feasible schedule for $(G[U \cup V], m, D_0)$. Moreover, it is easy to see that for any feasible schedule $S = (S_0, \dots, S_{\ell-1})$ for (G, m, D_0) , the time slot S_t is available with respect to $(\bigcup_{i=0}^{t-1} S_i, S_{t-1})$ for all $t \in \{0, \dots, \ell - 1\}$.

We will represent a partial schedule S for (G, m, D_0) by a tuple (U, V, t, ℓ) : U is the prefix of G , such that S is a feasible schedule for $(G[U], m, D_0)$, t is a starting time that exceeds the starting times of all tasks in U , V is the set of sinks of $G[U]$ that finish at time t and ℓ is the maximum tardiness of a task in U . Note that V may be empty. The time t is used to denote the next time at which the remaining tasks of G can be scheduled.

A tuple (U, V, t, ℓ) will be called a *feasible tuple* of (G, m, D_0) if U is a prefix of G , V is a set of sinks of $G[U]$, and there is a feasible schedule S for $(G[U], m, D_0)$ with tardiness ℓ , such that $S(u) \leq t - 1$ for all tasks u in U and $S(u) = t - 1$ for all tasks u in V . Since there are minimum-tardiness schedules for (G, m, D_0) of length at most n , we will only consider feasible tuples (U, V, t, ℓ) of (G, m, D_0) , such that $0 \leq t \leq n - 1$.

Let $S = (S_0, \dots, S_{\ell-1})$ be a feasible schedule for (G, m, D_0) . For each time $t \in \{0, \dots, \ell - 1\}$, the partial schedule (S_0, \dots, S_{t-1}) can be represented by the feasible tuple $(\bigcup_{i=0}^{t-1} S_i, S_{t-1}, t, \ell_t)$ of (G, m, D_0) , where $\ell_t = \max\{0, \max\{S(u) + 1 - D_0(u) \mid S(u) \leq t - 1\}\}$.

Note that a feasible tuple (U, V, t, ℓ) of (G, m, D_0) may represent more than one partial schedule. For all partial schedules S represented by (U, V, t, ℓ) , the availability of a set of sources of $G[V(G) \setminus U]$ at time t only depends on U and V . So all partial schedules represented by (U, V, t, ℓ) can be extended in the same way. Because the tardiness of such an extension only depends on ℓ and the starting times of the tasks of $G[V(G) \setminus U]$, the minimum-tardiness extensions of the schedules represented by (U, V, t, ℓ) all have the same tardiness. So to construct a minimum-tardiness schedule for (G, m, D_0) , we only need to consider feasible tuples of (G, m, D_0) .

Partial schedules for (G, m, D_0) can be extended by adding a time slot. The notion of extensions is used for feasible tuples as well. Let (U, V, t, ℓ) and (U', V', t', ℓ') be two feasible tuples of (G, m, D_0) . Then (U', V', t', ℓ') is called *available* with respect to (U, V, t, ℓ) if

1. $U' = U \cup V'$;

2. $t' = t + 1$; and
3. $\ell' = \max\{\ell, \max_{u \in V'}(t + 1 - D_0(u))\}$.

The set $Av(U, V, t, \ell)$ contains all feasible tuples of (G, m, D_0) that are available with respect to (U, V, t, ℓ) . Note that $Av(U, V, t, \ell)$ cannot be empty, because $(U, \emptyset, t + 1, \ell)$ is an element of $Av(U, V, t, \ell)$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) .

Let $S = (S_0, \dots, S_{\ell-1})$ be a feasible tuple of (G, m, D_0) . Then the feasible tuple $(\bigcup_{i=0}^{\ell} S_i, S_{\ell}, t + 1, \max\{0, \max\{S(u) + 1 - D_0(u) \mid S(u) \leq t\}\})$ of (G, m, D_0) is available with respect to the feasible tuple $(\bigcup_{i=0}^{\ell-1} S_i, S_{\ell-1}, t, \max\{0, \max\{S(u) + 1 - D_0(u) \mid S(u) \leq t - 1\}\})$ of (G, m, D_0) for all $t \in \{0, \dots, \ell - 1\}$.

Let (U, V, t, ℓ) be a feasible tuple of (G, m, D_0) . Assume S is a partial schedule for (G, m, D_0) corresponding to (U, V, t, ℓ) . Define $T(U, V, t, \ell)$ as the smallest tardiness of a feasible schedule for (G, m, D_0) that extends S . More precisely, if $U \neq V(G)$, then

$$T(U, V, t, \ell) = \min\{T(U', V', t', \ell') \mid (U', V', t', \ell') \in Av(U, V, t, \ell)\},$$

and if $U = V(G)$, then

$$T(U, V, t, \ell) = \ell.$$

Then $T(\emptyset, \emptyset, 0, 0)$ equals the tardiness of a minimum-tardiness schedule for (G, m, D_0) . Note that $T(U, V, t, \ell)$ is independent of the partial schedule corresponding to (U, V, t, ℓ) : each schedule S for $(G[U], m, D_0)$ with tardiness ℓ , such that $S(u) = t - 1$ for all tasks u in V and $S(u) \leq t - 1$ for all tasks u in U , can be extended to a feasible schedule for (G, m, D_0) with tardiness $T(U, V, t, \ell)$.

A minimum-tardiness schedule for (G, m, D_0) is computed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING presented in Figure 7.3. First, it computes a table Tab , such that $Tab[U, V, t, \ell]$ equals $T(U, V, t, \ell)$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) . Second, it uses this table to construct a minimum-tardiness schedule for (G, m, D_0) .

Now we will prove that Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING correctly constructs minimum-tardiness schedules.

Lemma 7.2.1. *Let S be the schedule for (G, m, D_0) constructed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. Then S is a minimum-tardiness schedule for (G, m, D_0) .*

Proof. Let Tab be the table constructed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. We will prove by induction that $Tab[U, V, t, \ell] = T(U, V, t, \ell)$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) . Let (U, V, t, ℓ) be a feasible tuple of (G, m, D_0) . Assume by induction that $Tab[U', V', t', \ell'] = T(U', V', t', \ell')$ for all feasible tuples (U', V', t', ℓ') in $Av(U, V, t, \ell)$.

If $U = V(G)$, then $T(U, V, t, \ell) = \ell$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) . In that case, $Tab[U, V, t, \ell] = T(U, V, t, \ell)$. So we may assume that $U \neq V(G)$. Because $T(U, V, t, \ell)$ equals $\min\{T(U', V', t', \ell') \mid (U', V', t', \ell') \in Av(U, V, t, \ell)\}$ and $Tab[U', V', t', \ell'] = T(U', V', t', \ell')$ for all feasible tuples (U', V', t', ℓ') in $Av(U, V, t, \ell)$, $Tab[U, V, t, \ell]$ equals $\min\{T(U', V', t', \ell') \mid$

Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING**Input.** An instance (G, m, D_0) .**Output.** A minimum-tardiness schedule for (G, m, D_0) .

```

1. for all feasible tuples  $(U, V, t, \ell)$  of  $(G, m, D_0)$ 
2.   do  $Tab[U, V, t, \ell] := \infty$ 
3. CONSTRUCT $(\emptyset, \emptyset, 0, 0)$ 
4.  $(U, V, t, \ell) := (\emptyset, \emptyset, 0, 0)$ 
5. while  $U \neq V(G)$ 
6.   do let  $(U', V', t', \ell') = succ(U, V, t, \ell)$ 
7.     for  $u \in V'$ 
8.       do  $S(u) := t$ 
9.          $(U, V, t, \ell) := (U', V', t', \ell')$ 
10.
11. Procedure CONSTRUCT $(U, V, t, \ell)$ 
12. if  $Tab[U, V, t, \ell] = \infty$ 
13.   then if  $U = V(G)$ 
14.     then  $Tab[U, V, t, \ell] := \ell$ 
15.     else  $T := \infty$ 
16.       for  $(U', V', t', \ell') \in Av(U, V, t, \ell)$ 
17.         do CONSTRUCT $(U', V', t', \ell')$ 
18.           if  $Tab[U', V', t', \ell'] < T$ 
19.             then  $T := Tab[U', V', t', \ell']$ 
20.                $succ(U, V, t, \ell) := (U', V', t', \ell')$ 
21.            $Tab[U, V, t, \ell] := T$ 

```

Figure 7.3. Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING

$(U', V', t', \ell') \in Av(U, V, t, \ell)\} = T(U, V, t, \ell)$. By induction, $Tab[U, V, t, \ell] = T(U, V, t, \ell)$ for all feasible tuples (U', V', t', ℓ') of (G, m, D_0) .

In addition, it is not difficult to see that for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) , if $U \neq V(G)$, then $succ(U, V, t, \ell)$ is a feasible tuple in $Av(U, V, t, \ell)$, such that $Tab[succ(U, V, t, \ell)] = Tab[U, V, t, \ell]$. Consequently, for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) , if $U \neq V(G)$, then $T(succ(U, V, t, \ell))$ equals $T(U, V, t, \ell)$.

Because $Tab[U, V, t, \ell]$ equals $T(U, V, t, \ell)$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) , $Tab[\emptyset, \emptyset, 0, 0]$ equals the tardiness of a minimum-tardiness schedule for (G, m, D_0) . This is used to construct a schedule for (G, m, D_0) . We inductively define feasible tuples (U_i, V_i, t_i, ℓ_i) of (G, m, D_0) . Let $(U_0, V_0, t_0, \ell_0) = (\emptyset, \emptyset, 0, 0)$. If $U_i \neq V(G)$, then let $(U_{i+1}, V_{i+1}, t_{i+1}, \ell_{i+1}) = succ(U_i, V_i, t_i, \ell_i)$. Assume (U_k, V_k, t_k, ℓ_k) is the last feasible tuple of (G, m, D_0) that can be constructed this way. Then $U_k = V(G)$. It is not difficult to prove that $T(U_i, V_i, t_i, \ell_i) = T(U_0, V_0, t_0, \ell_0)$ for all $i \in \{0, \dots, k\}$. So each feasible tuple (U_i, V_i, t_i, ℓ_i) of (G, m, D_0) represents a partial schedule for (G, m, D_0) that can be extended to a minimum-tardiness schedule for (G, m, D_0) . It is easy to prove by induction that the feasible tuple (U_i, V_i, t_i, ℓ_i) of (G, m, D_0) represents the partial schedule (V_1, \dots, V_i) for all $i \in \{0, \dots, k\}$. So (V_1, \dots, V_k) is a minimum-tardiness

schedule for (G, m, D_0) . This is the schedule constructed by Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING. \square

The time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING can be determined as follows. Consider an instance (G, m, D_0) , such that G is a precedence graph of width w . In order to obtain a better time complexity, we need to consider two possibilities depending on whether G is known to be a transitive reduction or not. If it is unknown whether G is a transitive reduction, then Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING should start by computing the transitive reduction of G . This takes $O(n^{2 \cdot 376})$ time [17]. In the remainder of the analysis of the time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING, we will assume that G is a transitive reduction.

Assume C_1, \dots, C_w is a chain decomposition of G , such that $C_i = \{c_{i,1}, \dots, c_{i,k_i}\}$ for all $i \in \{1, \dots, w\}$. From Lemma 7.1.4, such a chain decomposition can be constructed in $O(wn^2 + e^+ \sqrt{n})$ time.

Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING first computes $T(U, V, t, \ell)$ for all feasible tuples (U, V, t, ℓ) of (G, m, D_0) . Since there is a minimum-tardiness schedule for (G, m, D_0) of length at most n , we may assume that $t \in \{0, \dots, n-1\}$. In addition, because every task has at most n starting times, at most n^2 different values of ℓ need to be taken into account. A prefix U of G is a set $\bigcup_{i=1}^w \{c_{i,1}, \dots, c_{i,b_i}\}$, such that $0 \leq b_i \leq k_i$ for all $i \in \{1, \dots, w\}$. A set of sinks V of $G[U]$ is a subset of the set $\{c_{1,b_1}, \dots, c_{w,b_w}\}$. A subset V of $\{c_{1,b_1}, \dots, c_{w,b_w}\}$ can be represented by a tuple (a_1, \dots, a_w) , such that $a_i \in \{0, 1\}$ for all $i \in \{1, \dots, w\}$: $a_i = 1$ if $c_{i,b_i} \in V$ and $a_i = 0$ if $c_{i,b_i} \notin V$. So a feasible tuple of (G, m, D_0) can be represented by a tuple $(b_1, \dots, b_w, a_1, \dots, a_w, t, \ell)$, such that $0 \leq b_i \leq k_i$ and $a_i \in \{0, 1\}$ for all $i \in \{1, \dots, w\}$, $t \in \{0, \dots, n-1\}$ and $\ell \in \bigcup_{u \in V(G)} \{1 - D_0(u), \dots, n - D_0(u)\}$. So the number of feasible tuples of (G, m, D_0) is at most

$$n^3 2^w \prod_{i=1}^w (k_i + 1) \leq n^3 2^w \prod_{i=1}^w 2k_i \leq n^3 2^{2w} \prod_{i=1}^w \frac{n}{w} \leq 2^w n^{w+3}.$$

For every feasible tuple (U, V, t, ℓ) of (G, m, D_0) , Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING computes the set $Av(U, V, t, \ell)$. There is a one-to-one correspondence between the elements of $Av(U, V, t, \ell)$ and the sets of sources of $G[V(G) \setminus U]$. Because G is a precedence graph of width w and the sources of a precedence graph are incomparable, $G[V(G) \setminus U]$ has at most w sources. As a result, $Av(U, V, t, \ell)$ contains at most 2^w elements. Checking the availability of a tuple (U', V', t', ℓ') of (G, m, D_0) with respect to (U, V, t, ℓ) can be done as follows. U' must be the set $U \cup V'$, V' must be a set containing at most w sources of $G[V(G) \setminus U]$, every task in V may have at most one child in V' and every task in V' may have at most one parent in V . Because G is a transitive reduction, every task of G has indegree and outdegree at most w . So the availability of a set of sources of $G[V(G) \setminus U]$ can be checked in $O(w^2)$ time. Hence for each feasible tuple (U, V, t, ℓ) of (G, m, D_0) , Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING uses $O(w^2 2^w)$ time. So Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING constructs the table Tab in $O(w^2 2^{2w} n^{w+3})$ time.

It is not difficult to see that the construction of the minimum-tardiness schedule for (G, m, D_0) does not require as much time as the construction of the table. So Algorithm UNIT EXECUTION

TIMES DYNAMIC PROGRAMMING constructs a minimum-tardiness schedule for (G, m, D_0) in $O(w^2 2^{2w} n^{w+3})$ time. Hence we have proved the following result.

Theorem 7.2.2. *There is an algorithm with an $O(w^2 2^{2w} n^{w+3})$ time complexity that constructs minimum-tardiness schedules for instances (G, m, D_0) , such that G is a precedence graph of width w .*

Consequently, for constant w , a minimum-tardiness schedule for a precedence graph of width w can be constructed in polynomial time.

Theorem 7.2.3. *There is an algorithm with an $O(n^{w+3})$ time complexity that constructs minimum-tardiness schedules for instances (G, m, D_0) , such that G is a precedence graph of constant width w .*

Proof. Obvious from Theorem 7.2.2. □

7.3 An NP-completeness result

In the previous section, it was proved that there is a polynomial-time algorithm that constructs minimum-tardiness schedules for precedence graphs of bounded width with unit-length tasks on m processors. Moreover, using a generalisation of the algorithm presented in Chapter 6, a minimum-tardiness schedule for precedence graphs of width two with arbitrary task lengths can be constructed in polynomial time [91].

In this section, it will be shown that constructing a minimum-tardiness schedule for precedence graphs of width w on less than w processors is an NP-hard optimisation problem. This is proved using a polynomial reduction from PARTITION [33].

Problem. PARTITION

Instance. A set of positive integers $A = \{a_1, \dots, a_n\}$.

Question. Is there a subset A' of A , such that $\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a$?

PARTITION is a well-known NP-complete decision problem [33]. Let WIDTH3ON2 be the following decision problem.

Problem. WIDTH3ON2

Instance. An instance $(G, \mu, 2, D_0)$, such that G is a precedence graph of width three.

Question. Is there an in-time schedule for $(G, \mu, 2, D_0)$?

Using a polynomial reduction from PARTITION, it will be shown that WIDTH3ON2 is an NP-complete decision problem.

Lemma 7.3.1. *There is a polynomial reduction from PARTITION to WIDTH3ON2.*

Proof. Let $A = \{a_1, \dots, a_n\}$ be an instance of PARTITION. Define $N = \sum_{a \in A} a$ and $M = N + 1$. Construct an instance $(G, \mu, 2, D_0)$ as follows. G is a precedence graph consisting of three chains. The first two chains, C_1 and C_2 , each consist of $n + 1$ tasks $c_{1,i}$ and $c_{2,i}$ of length $\mu(c_{j,i}) = M$,

such that $c_{j,0} \prec_{G,0} \cdots \prec_{G,0} c_{j,n}$. The third chain, C_3 , consists of n tasks u_1, \dots, u_n with lengths $\mu(u_i) = a_i$ for all $i \in \{1, \dots, n\}$ and precedence constraints $u_1 \prec_{G,0} \cdots \prec_{G,0} u_n$. Let $D_0(u) = \frac{1}{2}N + (n+1)M$ for all tasks u of G . Now we can prove that there is a subset A_1 of A , such that $\sum_{a \in A_1} a = \sum_{a \in A \setminus A_1} a$ if and only if there is an in-time schedule for $(G, \mu, 2, D)$.

(\Rightarrow) Assume there is a subset A_1 of A , such that $\sum_{a \in A_1} a = \sum_{a \in A \setminus A_1} a$. Define $A_2 = A \setminus A_1$. A feasible in-time schedule S for $(G, \mu, 2, D_0)$ can be constructed as follows. For each $i \in \{1, \dots, n\}$ and $p \in \{1, 2\}$, if $a_i \in A_p$, then let

$$S(u_i) = iM + \sum_{j < i: a_j \in A_p} a_j.$$

Furthermore, for all $i \in \{0, \dots, n\}$, let

$$S(c_{1,i}) = iM + \sum_{j \leq i: a_j \in A_1} a_j \quad \text{and} \quad S(c_{2,i}) = iM + \sum_{j \leq i: a_j \in A_2} a_j.$$

Clearly, $S(c_{p,i+1}) \geq S(c_{p,i}) + M$ for all $i \in \{0, \dots, n\}$ and $p \in \{1, 2\}$. In addition, for all $i \in \{1, \dots, n\}$ and $p \in \{1, 2\}$, if $a_i \in A_p$, then

$$S(u_i) = S(c_{p,i-1}) + M \quad \text{and} \quad S(u_i) + \mu(u_i) = S(c_{p,i}).$$

So at most two tasks are executed at the same time. Furthermore, for all $i \in \{0, \dots, n-1\}$ and $p \in \{1, 2\}$, if $u_{i+1} \in A_p$, then

$$\begin{aligned} S(u_{i+1}) &= (i+1)M + \sum_{j < i+1: a_j \in A_p} a_j \\ &\geq iM + M \\ &> iM + a_i + \sum_{j < i} a_j \\ &\geq S(u_i) + \mu(u_i). \end{aligned}$$

So S is a feasible schedule for $(G, \mu, 2, D_0)$. Every task of G finishes at or before time $\max\{S(c_{1,n}) + \mu(c_{1,n}), S(c_{2,n}) + \mu(c_{2,n})\} = nM + \frac{1}{2}N + M = (n+1)M + \frac{1}{2}N$. So S is an in-time schedule for $(G, \mu, 2, D_0)$.

(\Leftarrow) Assume S is an in-time schedule for $(G, \mu, 2, D_0)$. Then all tasks of G are completed at or before time $(n+1)M + \frac{1}{2}N$. Let π be the processor assignment for S constructed by Algorithm PROCESSOR ASSIGNMENT COMPUTATION. Each processor can execute at most $n+1$ tasks in C_1 or C_2 , otherwise, S has length at least $(n+2)M > (n+1)M + \sum_{a \in A} a > (n+1)M + \frac{1}{2}N$. So both processors execute exactly $n+1$ tasks of length M . The sum of the execution lengths of all tasks of G equals $2(n+1)M + N$. So no processor is idle before time $(n+1)M + \frac{1}{2}N$. Define

$$A_1 = \{a_i \mid \pi(u_i) = 1\} \quad \text{and} \quad A_2 = \{a_i \mid \pi(u_i) = 2\}.$$

Since no processor is idle before time $(n+1)M + \frac{1}{2}N$, $\sum_{a \in A_1} a = (n+1)M + \frac{1}{2}N - (n+1)M = \frac{1}{2} \sum_{a \in A} a$.

□

Lemma 7.3.1 shows that constructing minimum-tardiness schedules for precedence graph of width three on two processors is an NP-hard optimisation problem. It is easy to see that a similar proof can be used to show that constructing minimum-tardiness schedules for precedence graphs of width w on less than w processors is NP-hard as well.

Theorem 7.3.2. *Constructing minimum-tardiness schedules for instances (G, μ, m, D_0) , such that G is a precedence graph of constant width w and $2 \leq m < w$, is an NP-hard optimisation problem.*

7.4 Another dynamic programming algorithm

In Section 7.2, it was proved that minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time if all tasks have unit length. In Section 7.3, it is shown that constructing minimum-tardiness schedules for precedence graphs of width w with tasks of arbitrary length on less than w processors is an NP-hard optimisation problem. The complexity of constructing minimum-tardiness schedules for precedence graphs of width w with arbitrary task lengths on at least w processors remains open. Without communication delays, minimum-tardiness schedules for precedence graphs of width w on w processors can be constructed by a list scheduling algorithm (using any priority list). This is not true for scheduling subject to unit-length communication delays.

Example 7.4.1. Consider the instance $(G, 3, D_0)$ shown in Figure 7.4. Note that G is a precedence graph of width three. It is not difficult to see that $(G, 3, D_0)$ is consistent. Moreover, $(G, 3, D_0)$ can be converted into a pairwise consistent instance without decreasing any individual deadlines. Using the lst-list $(a_1, b_3, b_1, b_2, c_3, c_1, c_2, d_1)$, Algorithm LIST SCHEDULING constructs the schedule shown in Figure 7.5. This is not an in-time schedule for $(G, 3, D_0)$, because d_1 violates its deadline. In Figure 7.6, an in-time schedule for $(G, 3, D_0)$ is shown. This schedule can be constructed by Algorithm LIST SCHEDULING using lst-list $(a_1, b_1, b_2, b_3, c_3, c_1, c_2, d_1)$.

Example 7.4.1 shows that list scheduling does not construct minimum-tardiness schedules for precedence graphs of width w on w processors. In this section, it will be shown that a minimum-tardiness schedule for precedence graphs of width w with arbitrary task lengths on at least w processors can be constructed in polynomial time for each constant w . Like in Section 7.2, we will use a dynamic-programming approach that can be generalised to scheduling problems with other objective functions [91].

Let G be a precedence graph of width w . Consider an instance (G, μ, m, D_0) , such that $m \geq w$. In a feasible schedule S for (G, μ, m, D_0) , at most w tasks can be executed simultaneously. Hence any feasible schedule for (G, μ, ∞, D_0) is a feasible schedule for (G, μ, m, D_0) as well. On the other hand, any feasible schedule for (G, μ, m, D_0) is also a feasible schedule for (G, μ, ∞, D_0) . Therefore we will consider instances (G, μ, ∞, D_0) .

A schedule S for (G, μ, ∞, D_0) is called *greedy* if for all tasks u of G , there is no feasible schedule S' for (G, μ, ∞, D_0) , such that $S'(u) < S(u)$ and $S'(v) = S(v)$ for all tasks $v \neq u$ of G .

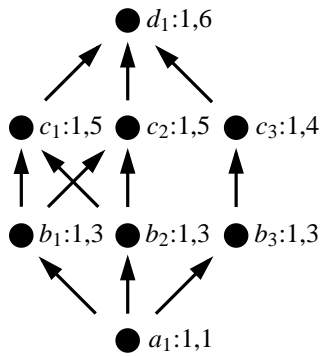


Figure 7.4. A consistent instance $(G, 3, D)$

0	1	2	3	4	5	6	7
a_1	b_3	c_3					
		b_1		c_1		d_1	
		b_2		c_2			

Figure 7.5. The schedule for $(G, 3, D)$ constructed by Algorithm LIST SCHEDULING

0	1	2	3	4	5	6	7
a_1	b_1			c_2	d_1		
		b_2	c_1				
		b_3	c_3				

Figure 7.6. An in-time schedule for $(G, 3, D)$

Note that the schedules for (G, μ, ∞, D_0) constructed by Algorithm LIST SCHEDULING are greedy schedules.

Let S be a feasible schedule for (G, μ, ∞, D_0) . Then S be transformed into a greedy schedule for (G, μ, ∞, D_0) as follows. Let u be a task of G . If u is available at time $t < S(u)$ and u can be scheduled at time t without violating the feasibility of S , then schedule u at time t . This is repeated until no task can be executed at an earlier time without violating the feasibility. The resulting schedule is a greedy schedule for (G, μ, ∞, D_0) . Since no task is scheduled at a later time, the tardiness of this schedule is at most that of S . Hence there is a greedy minimum-tardiness schedule for (G, μ, ∞, D_0) .

In a greedy schedule for (G, μ, ∞, D_0) , the number of potential starting times of a task is bounded. Let $est(u)$ denote the earliest possible starting time of a task u in a communication-free

schedule for (G, μ, ∞, D_0) .

$$est(u) = \begin{cases} 0 & \text{if } u \text{ is a source of } G \\ \max_{v \in Pred_{G,0}(u)} (est(v) + \mu(v)) & \text{otherwise} \end{cases}$$

In a greedy schedule for (G, μ, ∞, D_0) , every task u of G starts at most $n - 1$ time units after $est(u)$.

Lemma 7.4.2. *Let S be a feasible greedy schedule for (G, μ, ∞, D_0) . Then for all tasks u of G , $est(u) \leq S(u) \leq est(u) + n - 1$.*

Proof. Obviously, $S(u) \geq est(u)$ for all tasks u of G . For all tasks u of G , let $lpp(u)$ be the maximum number of tasks on a path from a source of G to a parent of u .

$$lpp(u) = \begin{cases} 0 & \text{if } u \text{ is a source of } G \\ \max_{v \in Pred_{G,0}(u)} lpp(v) + 1 & \text{otherwise} \end{cases}$$

We will prove by induction that $S(u) \leq est(u) + lpp(u)$ for all tasks u of G . This is obvious for the sources of G . Let u be a task of G . Assume by induction that $S(v) \leq est(v) + lpp(v)$ for all predecessors v of u . Let w be a predecessor of u with a maximum completion time. Then u is available at time $S(w) + \mu(w) + 1$. So u starts at time $S(w) + \mu(w)$ or at time $S(w) + \mu(w) + 1$. Consequently,

$$\begin{aligned} S(u) &\leq \max_{v \in Pred_{G,0}(u)} (S(v) + \mu(v) + 1) \\ &\leq \max_{v \in Pred_{G,0}(u)} (est(v) + lpp(v) + \mu(v) + 1) \\ &\leq \max_{v \in Pred_{G,0}(u)} (est(v) + \mu(v)) + \max_{v \in Pred_{G,0}(u)} (lpp(v) + 1) \\ &= est(u) + lpp(u). \end{aligned}$$

Clearly, $lpp(u) \leq n - 1$. So $est(u) \leq S(u) \leq est(u) + n - 1$. By induction, $est(u) \leq S(u) \leq est(u) + n - 1$ for all tasks u of G . \square

The limited number of potential starting times will be used in the design of a dynamic-programming algorithm. Let U be a prefix of G . Then any feasible schedule for $(G[U], \mu, \infty, D_0)$ can be extended to a feasible schedule for (G, μ, ∞, D_0) by assigning a starting time to the tasks of $G[V(G) \setminus U]$. This is the basis of the dynamic-programming algorithm.

Let S be a feasible schedule for $(G[U], \mu, \infty, D_0)$, such that $S(u) \leq t - 1$ for all tasks u in U . Let V be a set of sources of $G[V(G) \setminus U]$. Then V is called *available* at time t with respect to (U, S) if

1. for all tasks u in V , all parents of u are completed at or before time t ;
2. for all tasks u in V , at most one parent of u finishes at time t ; and
3. for all tasks u in U , if u finishes at time t , then V contains at most one child of u .

Note that the availability of V only depends on the completion times of the sinks of $G[U]$. Moreover, if S is a feasible schedule for (G, μ, ∞, D_0) , then for all times $t \in \{0, \dots, \max_{u \in V(G)}(S(u) + \mu(u))\}$, the set $\{u \in V(G) \mid S(u) = t\}$ is available at time t with respect to (U, S_U) , where $U = \{u \in V(G) \mid S(u) \leq t - 1\}$ and S_U is the restriction of S to U .

Partial (greedy) schedules for (G, μ, ∞, D_0) will be represented by tuples (U, S, t, ℓ) : t is an integer, such that $est(u) \leq t \leq est(u) + n - 1$ for some task u of G , U is a prefix of G and S is a schedule for $(G[U], \mu, \infty, D_0)$ with tardiness ℓ , such that $S(u) \leq t - 1$ for all tasks in U . The time t denotes the next time at which a task of G can be scheduled. Such a tuple (U, S, t, ℓ) will be called a *feasible tuple* of (G, μ, ∞, D_0) .

Since partial (greedy) schedules for (G, μ, ∞, D_0) can be extended by assigning a starting time to unscheduled tasks, we need a notion of extension of feasible tuples. Let (U, S, t, ℓ) and (U', S', t', ℓ') be two feasible tuples of (G, μ, ∞, D_0) . Then (U', S', t', ℓ') is called *available* with respect to (U, S, t, ℓ) if

1. $U' \setminus U$ is available at time t with respect to (U, S) ;
2. $t' \geq t + 1$; and
3. $\ell' = \max\{\ell, \max_{u \in U' \setminus U}(t + \mu(u) - D_0(u))\}$.

Let $Av(U, S, t, \ell)$ denote the set of feasible tuples of (G, μ, ∞, D_0) that are available with respect to (U, S, t, ℓ) . Note that if $U \neq V(G)$, then $Av(U, S, t, \ell)$ cannot be empty, since the feasible tuple (U, S, t', ℓ) , such that $t' = \min\{t'' \geq t + 1 \mid t'' \in \bigcup_{u \in V(G)}\{est(u), \dots, est(u) + n - 1\}\}$, is an element of $Av(U, S, t, \ell)$.

Let S be a greedy schedule for (G, μ, ∞, D_0) . Then for all times t , the tuple $(U_t, S_{U_t}, t, \ell_t)$, such that $U_t = \{u \in V(G) \mid S(u) \leq t - 1\}$, S_{U_t} is the restriction of S to U_t and ℓ_t is the tardiness of S_{U_t} , is a feasible tuple of (G, μ, ∞, D_0) . In addition, if $U_t \neq V(G)$, then the feasible tuple (U, S_U, t', ℓ_U) , where $t' = \min\{t'' \geq t + 1 \mid t'' \in \bigcup_{u \in V(G)}\{est(u), \dots, est(u) + n - 1\}\}$, $U = \{u \in V(G) \mid S(u) \leq t' - 1\}$, S_U is the restriction of S to U and ℓ_U is the tardiness of S_U , is available with respect to $(U_t, S_{U_t}, t, \ell_t)$. So to construct a minimum-tardiness schedule for (G, μ, ∞, D_0) , we only need to consider feasible tuples of (G, μ, ∞, D_0) .

Let (U, S, t, ℓ) be a feasible tuple of (G, μ, ∞, D_0) . Define $T(U, S, t, \ell)$ as the tardiness of a minimum-tardiness schedule for (G, μ, ∞, D_0) that extends S . Then for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) , if $U \neq V(G)$, then

$$T(U, S, t, \ell) = \min\{T(U', S', t', \ell') \mid (U', S', t', \ell') \in Av(U, S, t, \ell)\},$$

and if $U = V(G)$, then

$$T(U, S, t, \ell) = \ell.$$

Note that $T(\emptyset, \emptyset, 0, 0)$ equals the tardiness of a minimum-tardiness schedule for (G, μ, ∞, D_0) .

To implement the computation of $T(\emptyset, \emptyset, t, \ell)$, a table Tab is constructed. Tab contains an entry $Tab[U, S, t, \ell]$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) . We start by setting $Tab[U, S, t, \ell] = \infty$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) . Algorithm DYNAMIC PROGRAMMING presented in Figure 7.7 constructs a table Tab , such that $Tab[U, S, t, \ell] = T(U, S, t, \ell)$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) . This table is used to construct a minimum-tardiness schedule for (G, μ, ∞, D_0) .

Algorithm DYNAMIC PROGRAMMING

Input. An instance (G, μ, ∞, D_0) .

Output. A minimum-tardiness schedule for (G, μ, ∞, D_0) .

1. for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0)
2. do $Tab[U, S, t, \ell] := \infty$
3. CONSTRUCT($\emptyset, \emptyset, 0, 0$)
4. $(U, S, t, \ell) := (\emptyset, \emptyset, 0, 0)$
5. while $U \neq V(G)$
6. do let $(U', S', t', \ell') = succ(U, S, t, \ell)$
7. for $u \in U' \setminus U$
8. do $S(u) := t$
9. $(U, S, t, \ell) := (U', S', t', \ell')$
- 10.
11. **Procedure** CONSTRUCT(U, S, t, ℓ)
12. if $Tab[U, S, t, \ell] = \infty$
13. then if $U = V(G)$
14. then $Tab[U, S, t, \ell] := \ell$
15. else $T := \infty$
16. for $(U', S', t', \ell') \in Av(U, S, t, \ell)$
17. do CONSTRUCT(U', S', t', ℓ')
18. if $Tab[U', S', t', \ell'] < T$
19. then $T := Tab[U', S', t', \ell']$
20. $succ(U, S, t, \ell) := (U', S', t', \ell')$
21. $Tab[U, S, t, \ell] := T$

Figure 7.7. Algorithm DYNAMIC PROGRAMMING

Now we will prove that the schedules constructed by Algorithm DYNAMIC PROGRAMMING are minimum-tardiness schedules.

Lemma 7.4.3. *Let S be the schedule for (G, μ, ∞, D_0) constructed by Algorithm DYNAMIC PROGRAMMING. Then S is a minimum-tardiness schedule for (G, μ, ∞, D_0) .*

Proof. Let Tab be the table constructed by Algorithm DYNAMIC PROGRAMMING. We can prove by induction that $Tab[U, S, t, \ell]$ equals $T(U, S, t, \ell)$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) . Let (U, S, t, ℓ) be a feasible tuple of (G, μ, ∞, D_0) . Assume by induction that $Tab[U', S', t', \ell']$ equals $T(U', S', t', \ell')$ for all feasible tuples (U', S', t', ℓ') in $Av(U, S, t, \ell)$.

If $U = V(G)$, then $T(U, S, t, \ell) = \ell$. In that case, $Tab[U, S, t, \ell] = T(U, S, t, \ell)$. So we may assume that $U \neq V(G)$. Then $T(U, S, t, \ell)$ equals $\min\{T(U', S', t', \ell') \mid (U', S', t', \ell') \in Av(U, S, t, \ell)\}$. Algorithm DYNAMIC PROGRAMMING determines an element (U', S', t', ℓ') in $Av(U, S, t, \ell)$ with the smallest table entry. Hence $Tab[U, S, t, \ell] = T(U, S, t, \ell)$. By induction, $Tab[U, S, t, \ell] = T(U, S, t, \ell)$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) .

In addition, it is not difficult to see that for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) , if $U \neq V(G)$, then $succ(U, S, t, \ell) \in Av(U, S, t, \ell)$ and $Tab[succ(U, S, t, \ell)] = Tab[U, S, t, \ell]$. Since $Tab[U, S, t, \ell]$ equals $T(U, S, t, \ell)$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) , $Tab[\emptyset, \emptyset, 0, 0]$ equals the tardiness of a minimum-tardiness schedule for (G, μ, ∞, D_0) .

We inductively construct a sequence of feasible tuples (U_i, S_i, t_i, ℓ_i) of (G, μ, ∞, D_0) . Let $(U_0, S_0, t_0, \ell_0) = (\emptyset, \emptyset, 0, 0)$. If $U_i \neq V(G)$, then let $(U_{i+1}, S_{i+1}, t_{i+1}, \ell_{i+1}) = succ(U_i, S_i, t_i, \ell_i)$. Assume (U_k, S_k, t_k, ℓ_k) is the last feasible tuple that can be constructed this way. Then $U_k = V(G)$. Then the schedule S_k is the schedule for (G, μ, ∞, D_0) constructed by Algorithm DYNAMIC PROGRAMMING. S_k has tardiness ℓ_k . Because $T(U_k, S_k, t_k, \ell_k) = \ell_k = T(\emptyset, \emptyset, 0, 0)$ and $T(\emptyset, \emptyset, 0, 0)$ is the tardiness of a minimum-tardiness schedule for (G, μ, ∞, D_0) , Algorithm DYNAMIC PROGRAMMING constructs a minimum-tardiness schedule for (G, μ, ∞, D_0) . \square

The time complexity of Algorithm DYNAMIC PROGRAMMING can be determined as follows. Consider an instance (G, μ, ∞, D_0) , such that G is a precedence graph of width w . Like in the analysis of the time complexity of Algorithm UNIT EXECUTION TIMES DYNAMIC PROGRAMMING, we will assume that G is a transitive reduction.

Assume C_1, \dots, C_w is a chain decomposition of G , such that $C_i = \{c_{i,1}, \dots, c_{i,k_i}\}$ for all $i \in \{1, \dots, w\}$. From Lemma 7.1.4, C_1, \dots, C_w can be constructed in $O(wn^2 + e^+ \sqrt{n})$ time.

Algorithm DYNAMIC PROGRAMMING computes $T(U, S, t, \ell)$ for all feasible tuples (U, S, t, ℓ) of (G, μ, ∞, D_0) . There is a greedy minimum-tardiness schedule for (G, μ, ∞, D_0) . Hence we need to consider at most n^2 values of t and at most n^2 values of ℓ . A prefix U of G is a set $\bigcup_{i=1}^w \{c_{i,1}, \dots, c_{i,b_i}\}$, such that $0 \leq b_i \leq k_i$ for all $i \in \{1, \dots, w\}$. Because the availability of a feasible tuple with respect to (U, S, t, ℓ) only depends on the starting times of the sinks of $G[U]$, S can be represented by a tuple (t_1, \dots, t_w) , such that $t_i \in \bigcup_{i=1}^w \{est(c_{i,b_i}), \dots, est(c_{i,b_i}) + n - 1\}$ for all $i \in \{1, \dots, w\}$. So a feasible tuple (U, S, t, ℓ) of (G, μ, ∞, D_0) can be represented by a tuple $(b_1, \dots, b_w, t_1, \dots, t_w, t, \ell)$, such that $0 \leq b_i \leq k_i$ and $t_i \in \bigcup_{i=1}^w \{est(c_{i,b_i}), \dots, est(c_{i,b_i}) + n - 1\}$ for all $i \in \{1, \dots, w\}$, $t \in \bigcup_{u \in V(G)} \{est(u), \dots, est(u) + n - 1\}$ and $\ell \in \bigcup_{u \in V(G)} \{est(u) + \mu(u) - D_0(u), \dots, est(u) + n - 1 + \mu(u) - D_0(u)\}$. So the number of feasible tuples of (G, μ, ∞, D_0) is at most

$$n^4 \prod_{i=1}^w n(k_i + 1) \leq n^{w+4} \prod_{i=1}^w 2k_i \leq 2^w n^{w+4} \prod_{i=1}^w \frac{n}{w} \leq n^{2w+4}.$$

For each feasible tuple (U, S, t, ℓ) of (G, μ, ∞, D_0) , Algorithm DYNAMIC PROGRAMMING determines the set $Av(U, S, t, \ell)$. An element of $Av(U, S, t, \ell)$ corresponds to a subset of the sources of $G[V(G) \setminus U]$ and an integer t' , such that $est(u) \leq t' \leq est(u) + n - 1$ for some task u of G . Since G is a precedence graph of width w and the sources of a precedence graph are incomparable, $Av(U, S, t, \ell)$ contains at most $n^2 2^w$ elements. Since the availability of a feasible tuple

only depends on the starting times of the sinks and every task of G has indegree and outdegree at most w , checking whether a feasible tuple (U', S', t, ℓ) of (G, μ, ∞, D_0) is available with respect to (U, S) takes $O(w^2)$ time. Consequently, Algorithm DYNAMIC PROGRAMMING uses $O(n^2 w^2 2^w)$ time for each feasible tuple (U, S, t, ℓ) of (G, μ, ∞, D_0) . So the table Tab is constructed in $O(w^2 2^w n^{2w+6})$ time.

Using table Tab , Algorithm DYNAMIC PROGRAMMING constructs minimum-tardiness schedule for (G, μ, ∞, D_0) . It is obvious that the construction of the schedule does not take as much time as the construction of the table. As a result, Algorithm DYNAMIC PROGRAMMING constructs a minimum-tardiness for (G, μ, ∞, D_0) in $O(w^2 2^w n^{2w+6})$ time. Since any feasible schedule for (G, μ, ∞, D_0) is a feasible schedule for (G, μ, m, D_0) for all $m \geq w$, we have proved the following result.

Theorem 7.4.4. *There is an algorithm with an $O(w^2 2^w n^{2w+6})$ time complexity that constructs minimum-tardiness schedules for instances (G, μ, m, D_0) , such that G is a precedence graph of width w and $m \geq w$.*

For every fixed w , minimum-tardiness schedules can be constructed in polynomial time.

Theorem 7.4.5. *There is an algorithm with an $O(n^{2w+6})$ time complexity that constructs minimum-tardiness schedules for instances (G, μ, m, D_0) , such that G is a precedence graph of constant width w and $m \geq w$.*

Proof. Obvious from Theorem 7.4.4. □

7.5 Concluding remarks

In this chapter, it is proved that minimum-tardiness schedules for precedence graphs of bounded width can be constructed in polynomial time. It is obvious that the dynamic-programming approaches presented in this chapter can be generalised in many ways. First of all, both algorithms can be generalised for scheduling with other objective functions [91]. The same is true for scheduling subject to $\{0, 1\}$ -communication delays and for scheduling with release dates and deadlines. Both generalisations do not increase the time complexity.

The dynamic-programming algorithm for scheduling precedence graphs with unit-length tasks can be generalised in other ways as well. For instance, if a task cannot be executed by every processor or the communication delays may have length at least two, then there is a minimum-tardiness schedule whose length is bounded by a polynomial in the number of tasks. Consequently, the dynamic-programming algorithm presented in Section 7.2 can be generalised to a polynomial-time algorithm for such problems. This is not true for the algorithm presented in Section 7.4. This algorithm does not construct minimum-tardiness schedules for precedence graphs of bounded width in polynomial time if the number of possible starting times in a minimum-tardiness schedule is not bounded by a polynomial in the number of tasks. So this algorithm cannot be used for scheduling preallocated tasks. In addition, Sotskov and Shakhlevich [83] proved that constructing a minimum-length schedule on three processors for a job shop with

three jobs is an NP-hard optimisation problem. Hence it is unlikely that there is a polynomial-time algorithm that constructs minimum-tardiness schedules for precedence graphs of constant width w with preallocated tasks on $m \geq w$ processors.