SIAM J. ALG. DISC. METH.
Vol. 6, No. 4, October 1985

014

[133]

PROFILE SCHEDULING OF OPPOSING FORESTS AND LEVEL ORDERS*

DANNY DOLEV† AND MANFRED K. WARMUTH‡

Abstract. The question of existence of a schedule of a given length for n unit length tasks on m identical processors subject to precedence constraints is known to be NP-complete [Ullman, J. Comput. System Sci., 10 (1976), pp. 384-393]. For a *fixed value* of m we present polynomial algorithms to find an optimal schedule for two families of precedence graphs: level orders and opposing forests. In the case of opposing forest our algorithm is a considerable improvement over the algorithm presented in [Garey et al., SIAM J. Alg. Disc. Meth., 4 (1983), pp. 72-93].

1. Introduction. The goal of deterministic scheduling is to obtain efficient algorithms under the assumption that all the information about the tasks to be scheduled is known in advance [Co76], [GL79]. One of the fundamental problems in deterministic scheduling is to schedule a collection of n partially ordered, unit length tasks on a number of identical processors. As in [GJ83], [DW84a], [DW84b] we allow the number of identical processors to vary with time. This is described by a sequence of natural numbers, called a *profile* specifying how many processors are available at each unit of time (time slot). The *breadth* m, of a profile is an upper bound on the number of processors available at any time. A profile is *straight* if the number of available processors is the same at any time.

A *schedule* for a given profile is a partitioning of all the tasks into a sequence of sets which does not violate the precedence constraints and the number of tasks in each set does not exceed the number of available processors specified by the profile for the corresponding time slot.

Various aspects of scheduling theory have been extensively studied in recent years [GL79] and many scheduling problems are known to be NP-complete [GJ79]. The first NP-completeness result on scheduling with precedence constraints was published by Ullman [Ul75]. He showed that the existence of a schedule of a given length on a straight profile for a collection of unit length tasks subjected to precedence constraints is NP-complete in case where the breadth of the profile is a variable of the problem, that is, the breadth of the profile is not bounded by a constant. This problem remains NP-complete even for precedence graphs of special forms [GJ83], [Ma81], [Wa81].

Polynomial algorithms have been developed only for a few special cases of scheduling unit length tasks with precedence constraints. The first polynomial algorithm was developed by Hu [Hu61]. It produces an optimal schedule for a straight profile of arbitrary breadth if the precedence graph is either an inforest or an outforest. Hu's algorithm produces a schedule according to the *Highest Level First* (HLF) strategy, meaning tasks of higher level are chosen over tasks of lower level and among tasks of the same level ties are broken arbitrarily. Restricted versions of HLF provide optimal schedules if the precedence graph is an interval order [PY79], [Ga81], or if the number of available processors is two [FK71], [CG72], [Ga82].

The major scheduling problem remaining open is whether the scheduling of an arbitrary graph is NP-complete or polynomial for fixed number ($m \geq 3$) of processors. In this paper we address two special cases of the above open problem. We utilize the

* Received by the editors August 2, 1982, and in revised form May 31, 1984.

† IBM Research Laboratory, San Jose, California 95193. Current address: Institute of Mathematics and Computer Science, Hebrew University, Jerusalem, Israel.

‡ Computer Science Department, University of California, Santa Cruz, California 95064.

results presented in [Wa81], [DW84a] to obtain polynomial algorithms for two families of precedence constraints (precedence graphs): level orders and opposing forests. A graph is a level order if each connected component is partitioned into some k levels $L_0, \cdots, L_{k-1}$ such that for every two tasks $x \in L_i$ and $y \in L_j$, where $i > j$, x precedes y. We present an algorithm for finding optimal schedules for this class that requires time and space $O(n^{m-1})$. An opposing forest [GJ83] is a graph composed of intrees and outtrees only. It is a generalization of the cases solvable by Hu's [Hu61] algorithm. Garey, et al., [GJ83] presented a polynomial algorithm for finding an optimal schedule in the case of opposing forest and straight profile of fixed breadth $m \geq 3$. Their algorithm costs $O(n^{m^2+2m-5} \log n)$ time and $O(n)$ space. The algorithm we presented for this case is bounded by $O(n^{2m-2} \log n)$ time and $O(n^{m-1})$ space. For the special case $m = 3$ there exist a linear algorithms to find an optimal schedule [DW84a], [GJ83].

Our polynomial algorithms are based on the reduction theorem, which is proved in § 3. The reduction theorem is another form of the elite theorem [DW84a]. It reduces the number of components we have to consider at each step of the algorithm to at most $m-1$ (the highest ones) and therefore enables us to obtain efficient algorithms.

Notice that if the breadth of the profile is a variable of the problem rather than fixed, then scheduling a level order or an opposing forest becomes NP-complete [GJ83], [Ma81], [Wa81]. Thus our algorithms are expected to have a high complexity (exponential in the breadth m). A similar case was published in [DW84b]. It was shown that scheduling a precedence graph of bounded height on a profile of fixed breadth is polynomial. For profiles of arbitrary breadth the problem is again NP-complete [LR78], even if there is an arbitrary number of processors in only one time slot and one processor in all other slots [Wa81], [DW84a].

In § 2 we present the main notions used in the rest of the paper. Section 3 contains the reduction theorem. In §§ 4 and 5 we present the polynomial algorithm for level orders and opposing forests, respectively.

2. Basic definitions and properties.

2.1. Graph definitions. A (*precedence*) *graph* G is a directed acyclic graph given as a tuple (V, E), where V is the set of n *vertices* (or *tasks*) and E the set of *edges* of G. A (*directed*) *path* π of length r in a precedence graph $G = (V, E)$ is a sequence of vertices $x_0, \cdots, x_r$ such that the edge (x_i, x_{i+1}), for $0 \leq i \leq r-1$, is in E. A precedence graph G specifies the precedence constraints between the vertices (tasks) of G. We assume that if a task x has to be executed before a task y, then there exists a (directed) path of positive length from x to y in G, that is, x is a *predecessor* of y, and y is a *successor* of x. In the case where the longest path from a vertex x to a vertex y is the edge (x, y), x is an *immediate predecessor* of y and y is an *immediate successor* of x. Vertices x and y are *incomparable*, if x is neither a predecessor nor a successor of y. A set of vertices is incomparable if for any two vertices x and y of the set, x and y are incomparable, that is, there is no path between any two distinct vertices of the set.

By $h(G)$ we mean the *height* of G, which is the length of the longest path in G. For a vertex $x \in G$ (i.e., $x \in V$) we denote by $h(x)$ the length of the longest path that starts at x. A vertex with no successors has zero height. Vertices with identical height are said to be at the same *level*. Observe that all vertices of the same level are incomparable.

The graph G' is a (*closed*) *subgraph* of G if every vertex of G' has the same successors in G' as it has in G. A vertex of G, is *initial* if it has no predecessors. Note that an initial vertex of G is not necessarily of maximum height in G. A set of t *highest initial vertices* of G is a subset of initial vertices containing the t highest ones. Ties

are resolved arbitrarily. If there are less than t initial vertices then the set consists of all of them.

Let R be a set of initial vertices of G; then $G-R$ is the closed subgraph of G obtained by removing all the vertices of R from G. Given two graphs $G=(V, E)$ and $G'=(V', E')$, then $G \cup G'$ denotes the graph $(V \cup V', E \cup E')$. The graph $G=(V, E)$ is composed of $\{G_1, \cdots, G_r\}$ if these closed subgraphs (called *components* of G) are a decomposition of G into its connected components, that is each closed subgraph is a connected graph and there are no edges between vertices of different components; therefore, $G=\cup_i G_i$.

An *inforest* (respectively *outforest*) is a graph in which each vertex has at most one immediate successor (respectively one immediate predecessor). Notice that outforest is composed of components, each of which has exactly one initial vertex and it consists of this vertex and all its successors. A component of an outforest is called *outtree* and similarly a component of an inforest is called *intree*.

In a *level order* graph each component has the following form: Every vertex of level i precedes all vertices of the component from all the levels below i. Note that all vertices of the same component of a level order that are at the same level are isomorphic. Thus, we can assume that such a component is given as a tuple specifying how many vertices are in each level of the component.

2.2. Profile definitions. We partition the time scale into *time slots* of length one. The time interval $[i-1, i)$ for $i \geq 1$ is the ith time slot. A *profile* is a sequence of positive integers specifying the number of identical processors that are available in each time slot. We shall interpret profile $M=(m_1, \cdots, m_d)$, where d is its length, to mean that for each slot i in $[0, d)$ there are m_i processors available.

The *breadth* of profile M is the upper bound on the number of processors that are available at any time slot of M. The profile of Table 2.1 has breadth 4. Throughout the paper we denote the breadth of the given profile with the letter m. We call a profile M *straight* if $m_i=m$, for all $1 \leq i \leq d$.

TABLE 2.1
A schedule for G fitting the profile $M=(2, 4, 2, 1, 1)$.

slot	1	2	3	4	5
P_1	4	1	3	6	
P_2	5	2	9		
P_3		7			
P_4		8			
m_i	2	4	2	1	1

2.3. Schedule definition. A *schedule* S for a precedence graph G is a sequence of sets $(S)_1, \cdots, (S)_k$ such that:

(i) the sets $(S)_i$, for $1 \leq i \leq k$, partition the vertices of G;

(ii) if $x \in (S)_i$ and $y \in (S)_j$, for $1 \leq i \leq j \leq k$, then there is no path from y to x.

The *length* of a schedule $\lambda(S)$ is the index of the last nonempty set in the sequence. A minimal length schedule is called *optimal*. The schedule S *fits* the profile M if the length of S is not greater than the length of the profile and the cardinality of $(S)_i$ is not greater than m_i. The set of tasks $(S)_i$ get executed in the ith time slot, that is $|(S)_i|$ of the m_i processors of slot i each execute a task of $(S)_i$ during the time interval $[i-1, i)$. Note that all the tasks have unit length, which corresponds to the length of a time slot. An example is given in Fig. 2.1 and Table 2.1. The ith slot of S, $1 \leq i \leq \lambda(S)$,

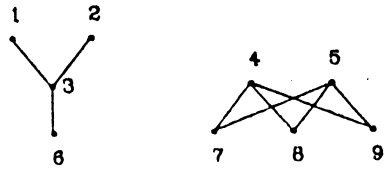

FIG. 2.1. *A precedence graph G.*

has $m_i - |(S)_i|$ *idle periods* meaning that there are this many processors idle during time slot i of S.

Given a precedence graph G and profile M, the initial problem is to determine if a schedule S exists for G and M. If a feasible schedule does exist, then we look for the shortest schedule S for G that fits M. In the first issue we allow the possibility that there does not exist a schedule for G that fits M. In the second we assume that there exists a feasible schedule and we are only interested in an optimal schedule.

A schedule S is an HLF-*schedule* for G and M if $(S)_i$, $1 \leq i \leq \lambda(S)$, is a set of m_i highest initial tasks of the closed subgraph of G induced by all tasks scheduled in slot i of S or later. HLF-schedules have the following property. Assume task x is scheduled in slot i and y is scheduled in slot j. If $h(x) > h(y)$, then either $i \leq j$ or there is a predecessor of x in the jth slot. We say that HLF *produces an optimal schedule* if any HLF-schedule is optimal; that is, if an optimal schedule can be constructed by choosing higher initial tasks before lower ones and choosing arbitrarily among initial tasks of the same height. Note that the schedule of Table 2.1 is not a HLF-schedule; moreover, no HLF-schedule is optimal for G (Fig. 2.1) and the profile of Table 2.1.

2.4. The median. The following definition relates the number of components of a graph and the heights of the components with m; where m is the breadth of the profile.

DEFINITION. The *median* of precedence graph G with respect to a given m, denoted by $\mu(G)$, is one plus the height of some mth highest component of G. If the graph has less than m components, then the median is 0.

For example, if the precedence graph is the one in Fig. 2.2 and the breadth of the given profile is three, then the median is three because three is one plus the height of the third highest component. For the graph described by Fig. 2.1 the median is 0 with respect to $m = 3$.

We use the median to split the precedence graph G into two subgraphs. Let $G = H(G) \cup L(G)$, where the *high-graph* $H(G)$ contains all components of G that are strictly higher than the median; the *low-graph* $L(G)$ is the remaining subgraph of G. Note that $H(G)$ has at most $m - 1$ components. Fig. 2.2 presents such a splitting of a precedence graph. We sometimes write $\mu(G, m)$, $H(G, m)$ and $L(G, m)$ to denote the median, the high-graph and the low-graph, respectively, for a specific m.

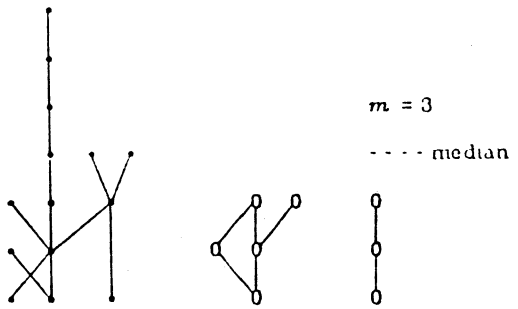

FIG. 2.2. *The decomposition of a graph G into H(G) and L(G); • denote vertices of H(G) and 0 vertices of L(G).*

The following properties of the median are used in the current paper.

PROPERTIES OF THE MEDIAN.

M1: There are at most $m-1$ components of G having height at least $\mu(G)$.

M2: If $\mu(G)>0$, then there are at least m components of G having height at least $\mu(G)-1$.

M3: If G has at most $m-1$ components of height at least h, then $\mu(G)\leqq h$.

M4: If G has at least m components of height at least $h-1$, then $\mu(G)\geqq h$.

The above properties follow directly from the definition of the median. Further properties of the median were given in [DW84a].

3. Reduction theorem. In this section we present our main result, the reduction theorem. We also prove several related theorems that are needed in later sections. The reduction theorem is a consequence of the MERGE Algorithm. The following lemma implies the correctness of the MERGE Algorithm. A component of a graph G is called *principal* if its height is at least $h(G)-1$.

LEMMA 3.1. *Let G be a graph and let G' be a subgraph of G obtained by removing a set of q highest initial tasks from G. Then G' contains at least as many principal components as the original graph G, unless $h(G')=0$.*

Proof. If the lemma holds for $q=1$, then it clearly holds for arbitrary q. Let x be a highest vertex of G, I be the principal component of G that contains x and $G'=G-\{x\}$. Assume $h(G')>0$. To show that G' contains at least as many principal components as G observe that $h(I)=h(G)>0$ and therefore $I-\{x\}$ contains a principal component of G'. Furthermore all principal components of G other than I are also principal components of G', because $h(G')\leqq h(G)$. We conclude that the number of principal components does not decrease when x is removed. □

The following algorithm shows how one can "merge" a schedule for a collection of subgraphs with a collection of subgraphs of lower height to get a schedule for the combined graph.

ALGORITHM 3.1. (the MERGE Algorithm)

Input: A graph $L=\bigcup_{i=1}^{r} L_i$ such that $h(L_i)\geqq h(L)-1$;
a graph $H=\bigcup_{i=1}^{q} H_i$ such that $h(H_i)>h(L)$, and $q+r\geqq m$;
a schedule S for H and M with p idle periods, where M is a profile of breadth m.

Output: A schedule S' for $H\cup L$ and M such that S' is not longer than the schedule S in the case where $p\geqq|L|$; and otherwise, S' is longer than S but has idle periods only in its last slot.

1. $k:=0$
   $S':=S$
2. **While** $h(L)>0$ **do**
   2.1. $k:=k+1$
   2.2. **While** $(S')_k$ is not full and not all initial vertices of H are scheduled in $(S')_k$ **do**
      2.2.1. Transfer an initial vertex of H from a slot after k to $(S')_k$.
   2.3. Fill $(S')_k$ with $m_k-|(S')_k|$ highest initial vertices of L.
   2.4. Remove the vertices of $(S')_k$ from L, H and its subgraphs H_i.
   2.5. **While** there is a subgraph H_i of H, such that $h(H_i)=h(L)$ **do**
      2.5.1. Transfer the graph H_i from H to L.
         $q:=q-1$; $r:=r+1$
      2.5.2. Remove the vertices of H_i from S'.

3. **While** L is nonempty **do**
   3.1. $k := k+1$
   3.2. **While** $(S')_k$ is not full or L is not empty **do**
      3.2.1. Add a vertex of L to slot k of S' and remove it from L.

A high level description of the MERGE *algorithm.* The aim of the algorithm is to "merge" the schedule S for H and M with the vertices of L producing a schedule S' for $H \cup L$ and M. The length of S' depends on the relationship between p, the number of idle periods in S and the number of vertices in L. If $p > |L|$, then there is enough "space" in S for all vertices of L and the resulting schedule S' is at most as long as S. Otherwise, S does not have enough idle periods and S' is longer than S. In this case, S' only has idle periods in its last slot.

At Step 1 of the algorithm we initialize S' with the schedule S for H and M. During Steps 2 and 3 the vertices of L are added into in S'. While doing so we sometimes reschedule vertices of H in S' (see Steps 2.2.1 and 2.5.2).

If $h(L) = 0$, then "merging" is easy (see Step 3). In this case, L is a set of single vertices. The algorithm consecutively fills the slots of S' with vertices of L until L is empty.

If $h(L) > 0$, then "merging" is slightly more involved (see Step 2). The variable q will be the number of subgraphs H_i that are left in H. All of these graphs will have height bigger than $h(L)$. If some of them drop down to height $h(L)$ during Step 2.4 then these subgraphs are transferred from H to L at Step 2.5. The variable r has the following meaning. During the algorithm it will be assumed that L has at least r principal components. The sum of q and r is at least m throughout the loop 2. This assures that there will be at least m initial vertices in $H \cup L$, at least q in H and at least r in L. We transfer components from H to L to avoid that some subgraphs H_i of H get completely scheduled and the sum of q and r drops below m.

Correctness of the MERGE *algorithm*: In the new schedule S' the precedence constraints specified by G are not violated, because we iteratively add vertices to S' (Steps 2.2.1, 2.3 and 3.2) that are initial in the unscheduled portion of $H \cup L$. Loop 2 has the following invariant: L has at least r principal components and H has q subgraphs H_i of height bigger than $h(L)$ and $q + r \geq m$.

Note that by the definition of H, L, q and r the loop invariant trivially holds after Step 1. We want to show that if the loop invariant holds before Step 2.1 and $h(L)$ is bigger than zero then it holds after Step 2.5, or $h(L)$ equals zero.

At Step 2.4 only initial vertices are removed from H, H_i and L. Therefore, their height can drop at most by one. This assures that after Step 2.4 the graph H contains q subgraphs H_i of height at least $h(L)$. By Lemma 3.2 we know that after Step 2.4 either the graph L contains at least r principal components or $h(L) = 0$. Note that at Step 2.3 $(S')_k$ was filled with highest initial vertices of L. At Step 2.5 all subgraphs H_i of H that dropped down to height $h(L)$ are transferred from H to L. The height $h(L)$ and the sum $q + r$ does not change during Step 2.5. Furthermore, if before Step 2.5.1 L has at least r principal components then L has also at least r principal components after Step 2.5.1, since each H_i that is transferred contains at least one component of height $h(L)$. This completes the proof of the invariant of Loop 2.

The following claim completes the proof of correctness. It shows that if $p \geq |L|$ then $\lambda(S') \leq \lambda(S)$, and if $p < |L|$ then $\lambda(S') > \lambda(S)$ and S' has idle periods only in its last slot.

CLAIM 3.1.

(i) *After Step* 3 *the schedule* $(S')_1, \cdots, (S')_{k-1}$ *does not have any idle periods.*

(ii) *After Step 3 either* $k = \lambda(S')$ *or* $\lambda(S') \leqq \lambda(S)$.

(iii) *If* $\lambda(S') > \lambda(S)$ *then* S' *can have idle periods only in its last slot.*

(iv) $p \geqq |L|$ *if and only if* $\lambda(S') \leqq \lambda(S)$.

Proof of (i). By the loop invariant we know that the current slot is filled up in Steps 2.2 and 2.3. Thus, the schedule $(S')_1, \cdots, (S')_k$ does not have any idle periods when Step 3 is reached. At Step 3 all the slots, except may be the last one, are completely filled. This completes the proof of (i).

Proof of (ii). At Step 1 the schedule S' is initialized with S, and therefore $\lambda(S') = \lambda(S)$. During Steps 2 and 3 the algorithm never adds any vertices to any slot of S' with a higher index than the current slot k. On the other hand, in Steps 2.2.1 and 2.5.2 there are vertices removed out of slots with higher indices than the current slot k. This implies that after Step 3, $k = \lambda(S')$ or $\lambda(S') \leqq \lambda(S)$.

Proof of (iii). If $\lambda(S') > \lambda(S)$ then (ii) implies that $k = \lambda(S')$ after Step 3. Applying (i) we get that S' can have idle periods only in its last slot.

Proof of (iv). Assume $\lambda(S') > \lambda(S)$; then by (iii) we know that S' can have idle periods only in its last slot. In particular, there are no idle periods in slots 1 through $\lambda(S)$ of S', which implies that $\sum_{i=1}^{\lambda(S)} m_i < |H| + |L|$. Since p can be expressed as $(\sum_{i=1}^{\lambda(S)} m_i) - |H|$, it follows that $p < |L|$.

To prove the opposite direction of (iv) assume that $p < |L|$. Expressing p as $\sum_{i=1}^{\lambda(S)} m_i - |H|$ implies that $\sum_{i=1}^{\lambda(S)} m_i < |H| + |L|$. Since S' is a schedule for $H \cup L$, we have $|H| + |L| \leqq \sum_{i=1}^{\lambda(S')} m_i$. Combining both inequalities we get $\sum_{i=1}^{\lambda(S)} m_i < \sum_{i=1}^{\lambda(S')} m_i$, which implies $\lambda(S) < \lambda(S')$.

Herewith we completed the proof of the claim and the proof of correctness of the MERGE algorithm. □

The MERGE algorithm is linear even if G is not transitively reduced [AH74].

LEMMA 3.2 [Wa81]. *The* MERGE *algorithm can be implemented in time and space* $O(n+e)$, *where* n *is the number of vertices and* e *the number of edges in* $H \cup L$.

Proof. We only give a general idea of the implementation of the MERGE algorithm. A complete description appears in [Wa81]. We keep track of the set of current initial vertices of H and L; call these sets I_H and I_L, respectively. Whenever we remove vertices from these sets we add the vertices that become initial to the list.

In Step 2.2.1 we can choose any vertex of I_H that is not already in $(S')_k$. On the other hand, vertices of I_L should be scheduled according to their height (Step 2.3). Thus we need a data structure that will enable us to retrieve vertices from I_L efficiently. We represent I_L as an array of lists, where the entry $I_L(h)$ points to a linked list of all the initial vertices of height h (in arbitrary order); see [DW84a], [Wa81] for details. As shown in the proof of correctness there are always enough initial vertices in $I_L(h(L))$ and $I_L(h(L)-1)$ to fill $(S')_k$ in Step 2.3. Thus it is enough to pick vertices of the last and second to last nonempty list of I_L. This is the main reason for the fact that the MERGE algorithm can be implemented in $O(n+e)$ time. We do not have to do a complicated search to find highest vertices in Step 2.3.

For Step 2.5 we need to keep track of the heights of the subgraphs H_i. This is easy since during each iteration of the loop the height of a subgraph H_i can drop at most by one. To be able to transfer components easily we need to keep track of the vertices of each H_i and keep pointers from each vertex of G to all its occurrences in the data structures. This completes the summary of the proof. □

The reduction theorem is an immediate consequence of the following theorem, in which we apply the MERGE algorithm 3.1.

THEOREM 3.1. *Let* G *be a graph and* M *be a profile of breadth* m. *Given a schedule* S *for the high-graph of* G *and* M *that has* p *idle periods, then with the* MERGE *algorithm*

one can find a schedule S' for the whole graph G and M in time and space $O(n+e)$ that has the following form:

(i) *if $p \geq |L(G)|$ then S' is at most as long as S;*

(ii) *if $p < |L(G)|$ then S' is longer than S and has idle periods only in its last slot.*

Proof. We run the MERGE algorithm on the following input parameters:

H is the high-graph and L the low-graph of G;

q is the number of components of $H(G)$ and $H_1, \cdots, H_q$ are the components of $H(G)$;

$r = m - q$ and $L_1, \cdots, L_{r-1}$ are some $r-1$ principal components of $L(G)$;

L_r is the remaining subgraph of $L(G)$ after removing $L_1, \cdots, L_{r-1}$.

Note that $h(H_i) > h(L)$, for $1 \leq i \leq q$, since H consists of all components of G that have height higher than the median of G, and L consists of all components which are at most as high as the median. By property M1 of the median we know that $H(G)$ has less than m components, and therefore $q < m$. Note that $H(G)$ might be empty and $q = 0$. Property M2 says that G has at least m components of height $\mu(G, m) - 1$. This implies that $L_1, \cdots, L_r$ exist and that $h(L_i) \geq h(L(G)) - 1$, for $1 \leq i \leq r$. Note that $h(L(G)) \leq \mu(G)$. It is easy to see that the input parameters can be found in time $O(n+e)$. Using Lemma 3.2 the proof is completed. □

We are now ready to present the main result of this section, the reduction theorem. It shows that finding an optimal schedule for G and M reduces to finding an optimal schedule for $H(G)$ and M.

THEOREM 3.2 (the reduction theorem). *Let G be a graph and M be a profile of breadth m. Then given an optimal schedule for the high-graph of G and M, the* MERGE *algorithm finds an optimal schedule for the whole graph G and M in time and space $O(n+e)$.*

Proof. Let S be the given optimal schedule for $H(G)$ and M. In Theorem 3.1 we showed that with the MERGE algorithm one can find a schedule S' for G and M in time and space $O(n+e)$ which has the following form:

(i) if $p \geq |L(G)|$, then $\lambda(S') \leq \lambda(S)$;

(ii) $p < |L(G)|$, then $\lambda(S') > \lambda(S)$ and S' has idle periods only in its last slot.

We want to show that the optimality of S for $H(G)$ and M implies the optimality of S' for G and M. Every schedule that has only idle periods in its last slot is optimal. Therefore, if $p < |L(G)|$ then S' is optimal. In the case $p \geq |L(G)|$, S' is at most as long as S. An optimal schedule for G and M has to be at least as long as an optimal schedule for $H(G)$ and M, since $H(G)$ is a closed subgraph of G. Thus in the case $p \geq |L(G)|$ we get $\lambda(S') = \lambda(S)$ and the optimality of S implies the optimality of S'. □

The following corollary of the reduction theorem implies that in the case where $H(G)$ is empty finding an optimal schedule is linear.

COROLLARY 3.1. *If $H(G)$ is empty then* HLF *is optimal for G and M and an* HLF *schedule can be found in time and space $O(n+e)$.*

Proof. The "empty schedule" is an optimal schedule for the empty graph $H(G)$ and M. The MERGE algorithm (applied as in Theorem 3.1) produces an arbitrary HLF schedule. Such a schedule has idle periods only in its last slot and is therefore optimal. □

The fact that HLF is optimal in the case where $H(G)$ is empty is also implied by the elite theorem of [DW84a].

The following theorem shows that the length of an optimal schedule is determined by the high-graph and the cardinality of the low-graph. The structure of the low-graph is not important.

THEOREM 3.3. *Let G and I be graphs such that $H(G, m) = H(I, m)$ and $|L(G, m)| =$*