Race-Condition Detection in Parallel Computation with Semaphores

Philip N. Klein Hsueh-I Lu
and Robert H. B. Netzer

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-96-04
January 1996

Race-Condition Detection in Parallel Computation with Semaphores

Philip N. Klein Hsueh-I Lu Robert H.B. Netzer

{klein,hil,rn}@cs.brown.edu, Department of Computer Science, Brown University

Abstract

We address a problem arising in debugging parallel programs, detecting race conditions in programs using semaphores for synchronization. It is NP-complete to detect race conditions in programs that use many semaphores [10]. We show in this paper that it remains NP-complete even if the programs are allowed to use only two semaphores.

For the case of single semaphore, Lu *et al.* [8] give the previously only-known polynomial-time algorithm that runs in time $O(n^{1.5}p)$, where p is the number of processors and n is the total number of semaphore operations executed. Their algorithm, however, detects only a special class of race conditions. In this paper we cope with the general race-condition detection problem and give an $O(np \log n)$-time algorithm. The output of our algorithm is a compact representation from which one can determine in constant time whether a race condition exists between two given operations.

1 Introduction

Race-condition detection is a crucial aspect of developing and debugging shared-memory parallel programs. Explicit synchronization is usually added to such programs to coordinate access to shared data, and *race conditions* result when this synchronization does not force concurrent processes to access data in the expected order. One way to dynamically detect races in a program is to trace its execution and analyze the traces afterward. A central part of dynamic race detection is to compute from the trace the order in which shared-memory accesses were *guaranteed* by the execution's synchronization to have executed. Accesses to the same location not guaranteed to execute in some particular order are considered a race. When programs use semaphore operations for synchronization,[1] some operations (belonging to different processes) could have potentially executed in an order different than what was traced. In this paper, we present fast algorithms for computing the order in which an execution's semaphore operations *could have* executed for the only case when it is tractable: programs that use a single semaphore. Our algorithms can be used to exactly detect race conditions in executions of such programs.

Past work has shown that exactly detecting races in programs that use multiple semaphores is NP-complete [10], and has developed exact algorithms for other cases where the problem is efficiently solvable (programs that use types of synchronization weaker than semaphores) [6,9], and heuristics for the multiple semaphore case [4,7]. The complexity for the case of constant number of semaphores has been an open question. We give the following theorem to settle the open question.

Theorem 1 The race-condition detection problem for more than one semaphore is NP-complete.

For the case of one semaphore we give two fast algorithms. Our goal was to solve this problem as efficiently as possible, since parallel programs are typically long-running, and the resulting large traces must be analyzed quickly by our algorithms. Our first algorithm determines whether any two given semaphore operations could have executed in a different order than during execution (and can be used

[1] When using a semaphore, a V-operation increments the semaphore, and a P-operation waits until the semaphore is greater than zero and then decrements the semaphore. P-operations are typically used to wait (synchronize) until some condition is true (such as a shared buffer becoming non-empty), and V-operations typically signal that some condition is now true.

to detect whether a race exists between any two particular operations) and runs in time and space linear in the total number of semaphore operations. Our second algorithm answers this question for all pairs of operations (and can detect all races in the execution) and runs in $O(np \log n)$ time, where n is the number of semaphore operations and p is the number of processors.

1.1 Reduction to Scheduling

Computing the order in which an execution's semaphore operations could have executed requires solving a scheduling problem. To determine whether any two operations, v and w, were guaranteed to have executed in a fixed order, we must determine if some valid subschedule of the execution's operations exists in which w precedes v. A *valid schedule* is an interleaving of the p processes' semaphore operations that honors the semantics of semaphore-style synchronization; i.e., a linear ordering of the operations such that at each point in the ordering, the number of V operations is never exceeded by the number of P operations (meaning that the semaphore is always nonnegative). A *valid subschedule* is a prefix of a valid schedule. Then, if the trace indicates that v preceded w in the actual execution, but a valid subschedule exists in which w precedes v, then v and w could have executed in either order. We consider subschedule because deadlocks might happen during the execution of parallel programs.

Determining whether a valid schedule exists in which w precedes v is equivalent to a special case of *sequencing to minimize maximum cumulative cost (SMMCC)*. We first describe the SMMCC problem and then explain the equivalence in the next two paragraphs. Given an acyclic directed graph G with costs on the nodes, a schedule is a topological ordering of the nodes; i.e. an ordering of the nodes consistent with the arcs. The cumulative cost of the first i nodes of such a schedule is just the sum of the cost of these nodes. Thus minimizing the maximum cumulative cost is an attempt to make sure that the cumulative cost stays low throughout the schedule. The SMMCC problem is NP-complete in general even if the node costs are restricted to ± 1 [1,5]. Abdel-Wahab and Kameda present an $O(n^2)$-time algorithm for the special case where G is a series-parallel graph [2] (the time bound was later improved to $O(n \log n)$ [3]). As part of this solution, they give an $O(n \log p)$-time algorithm applicable when G is a chain graph, a graph consisting of a union of p disjoint directed paths.

The existence problem of a valid *schedule* in which v precedes w can be reduced to the SMMCC problem in a chain graph augmented with one inter-chain edge. We add an edge from w to v, assign costs to the nodes (-1 if the node is a P-operation, $+1$ if a V-operation), and compute the minimum maximum cumulative cost. Clearly the cost is zero if and only if there is a valid schedule. The augmented chain graph is not series-parallel, so the algorithm of Abdel-Wahab and Kameda is not applicable. We show that the SMMCC problem can nevertheless be solved in polynomial time. In fact, for the special case of interest, that in which the costs are ± 1, we give a linear-time algorithm.

1.2 Single-Pair Race-Condition Detection

In order to detect race conditions between v and w, however, we need to find a valid *subschedule* containing v and w such that its maximum cumulative cost is minimized. Note that every valid subschedule of G is a valid schedule of a prefix subgraph of G. A graph G_0 is a *prefix subgraph* of G if (i) there is no arc of G from any node of $G - G_0$ to any node of G_0; (ii) every arc of G between two nodes of G_0 is also an arc of G_0. Clearly in the graph of interest, i.e. p parallel chains with an augmented arc, every prefix subgraph is determined by a cut comprising p cutpoints. Therefore the problem we address can be reduced to finding a cut such that the valid schedule of the prefix subgraph determined by the cut has the minimal maximum cumulative cost. Let h be the maximum cumulative cost of the optimal subschedule that contains v and w. If h is zero, then a valid subschedule exists (i.e. the optimal valid subschedule.) If h is positive, then there is no valid partial schedule because the maximum cumulative cost of any valid subschedule is greater than or equal to h and is thus positive, too. We shall show that a best cut can be found in linear time.

Theorem 2 Suppose G is a graph consisting of p disjoint chains comprising n nodes, where each node represents either a P-operation or a V-operation. For any two nodes v and w of G, one can determine in $O(n)$ time whether there is a valid subschedule in which v precedes w.

1.3 All-Pair Race-Condition Detection

In the application, parallel debugging, it is important to exactly detect all races. Hence we need to determine the above for all pairs of nodes v and w. Fortunately, there is a *compact representation* of this information. To represent this information, it is sufficient that we indicate, for each node v, and for each chain C not containing v, the first node w in C such that v precedes w in some valid subschedules. This representation has size $O(np)$, where n is the number of nodes and p is the number of chains.

The representation can be used to determine in constant time whether there is a race between two given operations v and w. A race exists if either operation can precede the other. To determine whether v can precede w, we obtain the first node in w's chain that could be preceded by v in some valid subschedules. If this first node is numbered later than w, then v can precede w. If not, v cannot precede w.

We therefore consider the complexity of constructing such a representation. Clearly it can be constructed by a sequence of calls to the algorithm of Theorem 2. We show how to do much better; in fact the time required by our algorithm is only $O(\log n)$ times the time required simply to write down the output.

Theorem 3 Suppose G is as in Theorem 2. The compact representation of the relation "v precedes w in some valid subschedules" can be constructed in $O(np \log n)$ time and $O(n)$ space.

1.4 Contribution

The previously best race-detection algorithm for one semaphore runs in time $O(n^{1.5}p)$ [8]. The positive result of this paper improves the time complexity by a factor of $\Omega(\sqrt{n}/\log n)$. Furthermore the race conditions considered in this paper is more general than that in [8], in which the race condition is considered only for valid schedules instead of valid subschedules. Specifically, in [8] v could precede w is defined to be the existence of a valid *schedule* (not subschedule) of G in which v precedes w. Therefore some race condition defined in this paper is not regarded as a race condition in [8]. Hence the algorithms in this paper are the first polynomial-time algorithms that detect general race conditions in programs that use semaphores.

The negative result of this paper shows that as long as a parallel programs uses more than one semaphore, detecting race condition, special or general, is NP-complete.

The rest of the paper is organized as follows. Section 2 gives some preliminary definitions and lemmas. The following three sections then prove Theorem 2, Theorem 3, and Theorem 1, respectively.

2 Preliminaries

2.1 Definition and Notation

Suppose G is an acyclic graph with node costs. We introduce some terminology having to do with schedules. Much of this terminology is adapted from that in [2]. A *schedule* of G is a sequence of G's nodes which is consistent with the precedence constraints imposed by the arcs of G. A *segment* of a schedule is a consecutive subsequence. Let $H = v_1 v_2 \cdots v_m$ be a sequence of nodes. The *cost* of H, denoted $c(H)$, is the sum of the costs of its nodes. The *height of a node v_ℓ in H* is defined to be the sum of the costs of the nodes v_1 through v_ℓ. The *height of H*, denoted $h(H)$, is the maximum height of any node in H. A node of maximum height in H is called a *peak*. A node of minimum height in H is called a *valley*. The *reverse height* of H, denote $\tilde{h}(H)$, is the height of H minus the cost of H. Note that height and reverse height are nonnegative. A schedule of G is *optimal* if its height is minimum over all schedules of G. We use $h(G)$ to denote the height of its optimal schedule.

A sequence $C = v_1 v_2 \cdots v_m$ of nodes of G is called a *chain* of G if the only edges in G incident on these nodes are $v_0 v_1, v_1 v_2, \ldots, v_{m-1} v_m, v_m v_{m+1}$, where v_0 and v_{m+1} are other nodes, denoted *pred*(C) and *succ*(C), respectively. We use *start*(C) to denote v_1 and *end*(C) to denote v_m. Note that C could be a single node.

We use $[v, w]_G$ to denote the chain of G starting from v and ending at w. Let $[v, -]_G$ denote the

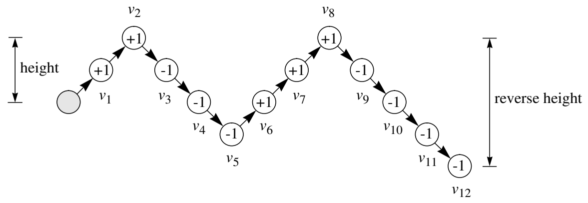


Figure 1: A hump H of 12 nodes: $v_1, v_2, \ldots, v_{12}$. The cost of each node is in the circle. By definition $c(H) = -2$, $h(H) = 2$, and $\tilde{h}(H) = 4$. Both of v_2 and v_8 are peaks of H, but only v_2 is useful.

longest chain of G starting from v, and $[-, v]_G$ the longest chain of G ending at v. If it is clear from the context which graph is intended, we may omit the subscript G. Note that the above notation might not be well-defined for any acyclic graph G, but it is so when G is composed of disjoint chains, which is the case of interest in this paper.

Suppose H is a chain of G containing a peak v_ℓ such that (1) every node of H preceding v_ℓ has nonnegative height in H, and (2) every node of H following v_ℓ has height in H at least the cost of H. In this case, we call H a *hump*, and we say v_ℓ is a *useful peak* of H. This definition is illustrated in Figure 1. We say a hump is an N*-hump* if its cost is negative, a P*-hump* if its cost is nonnegative.

We are concerned primarily with graphs consisting of the disjoint union of chains. For convenience, we assume in such a graph the existence of an *initial pseudonode* ($\perp$), preceding all nodes, and a *terminal pseudonode* ($\top$), following all nodes, each of cost zero. Thus *pred*(v) could be $\perp$ and *succ*(v) could be $\top$.

For the rest of the section we describe the properties of humps in schedules which will be used later. Most of them are adapted from [8].

2.2 Hump Decomposition

As part of their scheduling algorithm for series-parallel graphs, Abdel-Wahab and Kameda show that in linear time a sequence of nodes can be decomposed into a set of humps. The algorithm DECOMP() is shown in Figure 2. It takes a chain as input and outputs a set of disjoint subchains such that every subchain is a hump. The first Repeat-loop produces N-humps; and the second Repeat-loop produces P-humps. Each loop alternates between identifying peaks and valleys. It is not difficult to see that every sequence of nodes between two consecutive valleys is a hump. An example is shown in Figure 3. The chain is decomposed by DECOMP() into two N-humps and three P-humps. For a chain C, we say H is a *hump of C* if $H \in$ DECOMP(C). It can be proved that DECOMP() has the following properties.

```
Function DECOMP(C)
  S := {};
  u := the first valley of C;
  Repeat
    v := the first peak of [pred(C), u];
    w := the first valley of [pred(C), v];
    S := S ∪ {[succ(w), u]};
    u := w;
  Until u = pred(C);
  u := the first valley of C;
  Repeat
    v := the last peak of [u, end(C)];
    w := the last valley of [v, end(C)];
    S := S ∪ {[succ(u), w]};
    u := w;
  Until u = end(C);
  Return S;
```

Figure 2: The algorithm to decompose a chain into a set of humps

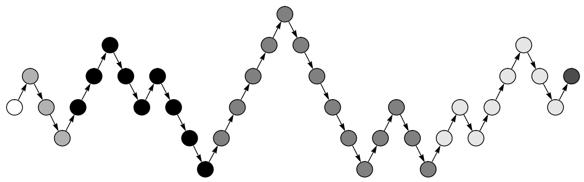


Figure 3: A chain decomposed into two N-humps and three P-humps

Hump-decomposition properties:

1. Suppose $H_1, H_2 \in$ DECOMP(C) and H_1 precedes H_2 in C. If $c(H_1) \geq 0$ then $c(H_2) \geq 0$ and $\tilde{h}(H_1) > \tilde{h}(H_2)$. If $c(H_2) < 0$ then $c(H_1) < 0$ and $h(H_1) < h(H_2)$.

2. If v is the first valley of $[u, w]$, then DECOMP$([u, v])$ is a set of N-humps, and

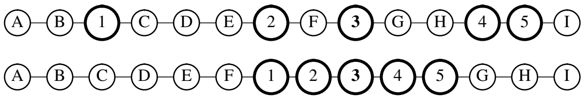


Figure 4: The second sequence of nodes is obtained from the first one by clustering the numbered nodes to node 3.

DECOMP$([succ(v), w])$ is a set of P-humps.

3. Let C and C' be two disjoint chains, whose humps are respectively $H_1, H_2, \ldots, H_k$ and $H_{k+1}, H_{k+2}, \ldots, H_\ell$ in order. Then for some $1 \le i \le k$ and $k \le j \le \ell$, the humps of CC' are $H_1, H_2, \ldots, H_i, (H_{i+1} \cdots H_j), H_{j+1}, \ldots, H_\ell$ in order.

The third property implies that

$$\begin{aligned}\{end(H) : H \in \text{DECOMP}(CC')\} \subseteq \\ \{end(H) : H \in \text{DECOMP}(C)\} \cup \\ \{end(H) : H \in \text{DECOMP}(C')\}.\end{aligned}$$

It will turn out that once we decompose a chain into humps, we need not be concerned with the internal structure of these humps. For each hump H we need only store $c(H)$ and $h(H)$. Thus a chain consisting of ℓ humps can be represented by a length-ℓ sequence of pairs $(c(H), h(H))$. We call this sequence the *hump representation* of the chain. Using the third hump-decomposition property, one could straightforwardly derive the hump representation of C_1C_2 from the hump representation of C_1 and that of C_2. In particular, if we are given DECOMP(C) and DECOMP(C'), then computing DECOMP(CC') takes time linear in $|\text{DECOMP}(C)| + |\text{DECOMP}(C')|$.

2.3 Hump Clustering

The following lemma concerns an operation on a schedule called *clustering* the nodes of a hump. Suppose H is a hump of G, and let v be a useful peak of H. Let S be a schedule of G. If all the nodes of H are consecutive in S, we say H is *clustered in* S. If every hump of G is clustered in S, we say the schedule S is *clustered*. If a hump is not clustered in a schedule, we can modify the schedule to make it so. To *cluster the nodes of H to v* is to change the positions of nodes of H other than v so that all the nodes of H are consecutive, and the order among nodes of H is unchanged. An example is shown in Figure 4.

Lemma 1 Let G be an acyclic graph with node costs and H be a hump of G. Suppose S is a schedule of G. If T is obtained from S by clustering all nodes in H to a useful peak of H, then T is a schedule of G and $h(T) \le h(S)$.

An example is shown in Figure 5. The height of the schedule in (c) is smaller than that of the schedule in (b). It follows from Lemma 1 that there is always a clustered optimal schedule of G. Two clustered schedules of the graph in Figure 5-(a) are shown in Figure 5-(d) and (e). We prove the lemma as follows.

Proof of Lemma 1 The original lemma in [2] restricts G to be a chain graph. We can prove as follows that the same properties hold even without the restriction. Suppose $H = v_1 \cdots v_p \cdots v_d$, where v_p is a useful peak of H. Suppose $w_1 w_2 \cdots w_\ell$ is the segment of S such that $w_{j_i} = v_i$ for all $1 \le i \le d$ and $1 = j_1 < j_2 < \cdots < j_d = \ell$. The only difference between S and T is that the segment of $W = w_1 w_2 \cdots w_\ell$ in S is replaced with $W' = W'_1 H W'_2$ in T, where

$$\begin{aligned} W'_1 &= w_{j_1+1} \cdots w_{j_2-1} w_{j_2+1} \cdots w_{j_p-1} \\ W'_2 &= w_{j_p+1} \cdots w_{j_{p+1}-1} w_{j_{p+1}+1} \cdots w_{j_d-1}. \end{aligned}$$

Suppose w_j is not in H. By definition of chains the precedence relations between v_i and w_j imposed by G are the same over all $1 \le i \le d$. Note that w_j precedes some node of H in W and succeeds some other node of H in W. It follows that there is no precedence constraint between v_i and w_j. Therefore T is a schedule of G.

We denote the heights of w_i in S and in T by $h_S(w_i)$ and $h_T(w_i)$, respectively. Note that $h_S(v_p) = h_T(v_p)$ since the set of nodes preceding v_p does not change. We show that for every $1 \le \alpha \le \ell$ there exists a $1 \le \beta \le \ell$ such that $h_T(w_\alpha) \le h_S(w_\beta)$.

- If $\alpha = j_i$ for some $1 \le i \le d$, then $h_T(w_\alpha) = h_T(v_i) \le h_T(v_p) = h_S(v_p) = h_S(w_{j_p})$.

- If $j_i < \alpha < j_{i+1}$ for some $p \leq i < d$, then $h_T(w_\alpha) = c(v_{i+1}) + c(v_{i+2}) + \cdots + c(v_d) + h_S(w_\alpha)$. Since H is a hump and $i \geq p$, $c(v_{i+1}) + c(v_{i+2}) + \cdots + c(v_d) = h_S(v_d) - h_S(v_i) \leq 0$. Thus $h_T(w_\alpha) \leq h_S(w_\alpha)$.

- If $j_i < \alpha < j_{i+1}$ for some $1 \leq i < p$, then $h_T(w_\alpha) = -c(v_1) - c(v_2) - \cdots - c(v_i) + h_S(w_\alpha)$. Since H is a hump and $i < p$, $-c(v_1) - c(v_2) - \cdots - c(v_i) = h_S(v_0) - h_S(v_i) \leq 0$, where v_0 is the node that precedes v_1 in S. Thus $h_T(w_\alpha) \leq h_S(w_\alpha)$.

It follows that $h(W') \leq h(W)$. Since nodes other than the w_i's preserve their heights in S and T, the lemma is proved. $\square$

It follows from Lemma 1 that there is always an optimal schedule of G which is clustered.

2.4 Standard Order

Consider a series $S_1 \ldots S_m$ of subsequences of nodes. It is in *standard order* if it satisfies the following properties:

Standard order properties:

- The series consists of S_i's with negative costs, followed by S_i's with nonnegative costs;

- The S_i's with negative costs are in nondecreasing order of height; and the S_i's with nonnegative costs are in nonincreasing order of reverse height.

If the humps of a chain are $H_1, H_2, \ldots, H_m$ in order, then the series $H_1 H_2 \cdots H_m$ is in standard order by the first hump-decomposition property.

Lemma 2 Let A, B, S_1 and S_2 be subsequences of nodes. Suppose $S = S_1 A B S_2$ and $T = S_1 B A S_2$. If the series BA is in standard order then $h(S) \geq h(T)$.

For example, the sequence in Figure 5-(d) is a clustered schedule of the graph in Figure 5-(a). Note that the series of the last two humps in the schedule is not in standard order: the reverse height of the first hump (zero) is less than that of the second hump (one). The schedule in Figure 5-(e) obtained by exchanging those two clustered humps has height one less than that of the schedule in Figure 5-(d).

Proof of Lemma 2 The original version of this lemma in [2] restricts A, B to be humps G. We can prove as follows that the same property holds even without these restrictions. Since the heights of nodes in S_1 and S_2 are not changed in S and T, it suffices to ensure that

$$\begin{aligned} h(AB) &= \max\{h(A), c(A) + h(B)\} \\ &\geq \max\{h(B), c(B) + h(A)\} \\ &= h(BA). \end{aligned}$$

- If $c(A) < 0$ and $c(B) < 0$, since the series BA is in standard order, $h(A) \geq h(B)$. Since $c(B) < 0$, it follows that $h(A) > c(B) + h(B)$.

- If $c(A) \geq 0$ and $c(B) \geq 0$, since the series BA is in standard order, $h(B) - c(B) = \tilde{h}(B) \geq \tilde{h}(A) = h(A) - c(A)$. Thus $c(A) + h(B) \geq c(B) + h(A)$. Since $c(A) \geq 0$, $c(A) + h(B) \geq h(B)$.

- If $c(A) \geq 0$ and $c(B) < 0$, then $h(A) > c(B) + h(A)$ and $c(A) + h(B) \geq h(B)$.

Since in all cases each of $h(B)$ and $c(B) + h(A)$ is less than or equal to one of $h(A)$ and $c(A) + h(B)$, the lemma is proved. $\square$

2.5 Hump Merging

A schedule of G is in *standard form* if it is clustered and its series of humps of G is in standard order. Let T be any schedule of G in standard form. Recall that by Lemma 1 there is always an optimal schedule S of G which is clustered. The humps of G, while clustered in both T and S, may not be in the same order. However, any two humps of the same chain of G must be in the same order in T and in S, else either T or S is not a schedule. Take two consecutive humps in S that are from different chains and that are not in the same order as in T, and exchange their positions. By Lemma 2 the resulting ordering has height no more than S. By a series of such exchanges, we eventually obtain T from S. It follows that the height of T is no more than that of S, and hence that T is optimal. This argument shows that every schedule in standard form is an optimal schedule of G.

Let $I = \{H_1, H_2, \ldots, H_m\}$, where the series $H_1 H_2 \cdots H_m$ is in standard order. Suppose

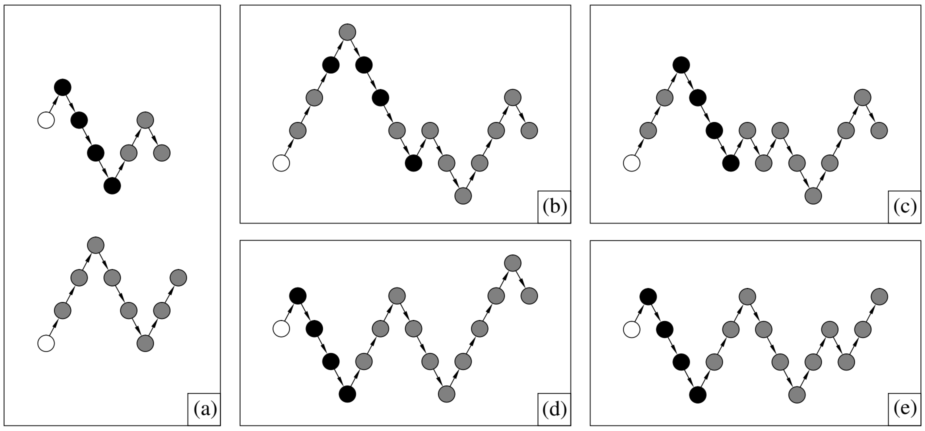


Figure 5: (a) The graph G is composed of two chains. The first chain is composed of an N-hump followed by a P-hump. The second chain is composed two P-humps. (b) A schedule for G of height four. (c) The schedule obtained from the previous one by clustering the N-hump to its useful peak. (d) A clustered schedule of G of height two. This one is obtained from the previous schedule by clustering every hump. (e) A clustered schedule of G of minimum height.

MERGE(I) returns a sequence of nodes obtained by concatenating all humps in I into standard order. Namely, MERGE$(I) = H_1 H_2 \cdots H_m$. Assume for uniqueness that MERGE() breaks ties in some arbitrary but fixed way. By the above argument we have the following lemma.

Lemma 3 If G is composed of disjoint chains, $C_1, C_2, \ldots, C_p$, then MERGE$(\bigcup_{1 \le i \le p}$ DECOMP$(C_i))$ is an optimal schedule of G.

An example is shown in Figure 5. Since the schedule in Figure 5-(e) is clustered and its series of humps is in standard order, it is an optimal schedule of the graph in Figure 5-(a). Abdel-Wahab and Kameda show that MERGE$(\bigcup_{1 \le i \le p}$ DECOMP$(C_i))$ can be obtained in $O(n \log p)$ time. Note that the output of function MERGE() may not be unique. Without loss of generality, however, we may define MERGE() more restrictively as follows to make its output unique for the same G. Suppose G is composed of disjoint chains, $C_1, C_2, \ldots, C_p$ and $I = \bigcup_{1 \le i \le p}$ DECOMP(C_i). Define MERGE$(I) = H_1 H_2 \cdots H_m$, where $\{H_1, H_2, \ldots, H_m\} = I$ and the series $H_1 H_2 \cdots H_m$ is in standard order. Furthermore, if $H_i H_j$ and $H_j H_i$ are both in standard order, where $C_{i'}$ contains H_i, $C_{j'}$ contains H_j, and $i' < j'$, then H_i precedes H_j in MERGE(I).

3 Algorithm for Single Pair

Let G be a graph composed of disjoint chains, $C_1, C_2, \ldots, C_p$. Recall that there are two pseudonodes, $\perp$ and $\top$. The cost of each node of G is either $+1$ or -1. A subschedule S of G is *valid* if $h(S) = 0$. Let v and w be two nodes of G. In this section we show how to determine in linear time whether v could precede w in some valid subschedule of G.

3.1 Notation

A vector $\Gamma = (x_1, x_2, \ldots, x_p)$ of p nodes is called a *cut* of G if each x_i is either $\perp$ or a node in C_i. We call x_i the i^{th} *cutpoint* of Γ. The prefix subgraph $G[\Gamma]$ of G is the subgraph $\cup_{1 \le i \le p}[-, x_i]$. Since we will frequently encounter two cuts that differ at only one cutpoint, let NEWCUT(Γ, i, u) denote a cut Γ' defined by

$$\Gamma'(\ell) = \begin{cases} \Gamma(\ell) & \text{if } \ell \ne i \\ u & \text{if } \ell = i. \end{cases}$$

A *j-schedule* of $G[\Gamma]$ is a schedule of $G[\Gamma]$ whose last node is $\Gamma(j)$. We use $h_j(G[\Gamma])$ to denote the height of an optimal j-schedule of $G[\Gamma]$. Suppose $\Gamma(j)$ is not $\perp$. One can compute $h_j(G[\Gamma])$ for a given Γ follows. Let $\Gamma' =$ NEWCUT$(\Gamma, j, \mathit{pred}(\Gamma(j)))$. Clearly $S\Gamma(j)$ is an optimal j-schedule of $G[\Gamma]$ if S is an optimal schedule of $G[\Gamma']$. (However, the other direction is not true.) It follows that

$$h_j(G[\Gamma]) = \max\{h(G[\Gamma']), c(G[\Gamma']) + h(\Gamma(j))\}.$$

Note that $h(G[\Gamma])$ and $h_j(G[\Gamma])$ are both nonnegative.

We use $v \to w$ to signify that there is a valid subschedule of G in which v precedes w. Let $v \not\to w$ signify that $v \to w$ is not true. Note that $\to$ is not a partial order, and neither is $\not\to$.

3.2 Basic Idea

Every valid subschedule of G is a valid schedule of a prefix subgraph $G[\Gamma]$ for some cut Γ of G. Therefore $v \to w$ if and only if there is a cut Γ of G such that $G[\Gamma]$ has a valid schedule in which v precedes w. Let h^* be the minimum of $h(G[\Gamma] \cup \{vw\})$ over all $G[\Gamma]$'s that contain v and w. It follows that $w \to w$ if and only if $h^* = 0$. Hence the problem of determining whether $v \to w$ is reduced to computing the minimum height of a set of chain graphs each augmented with an interchain arc. Clearly two immediate questions arise. 1) How do we compute the height of $G[\Gamma] \cup \{vw\}$, which is not even serial-parallel? 2) How do we cope with the fact that there could be exponential number of prefix subgraphs that contain v and w?

Let v and w be contained in two disjoint chains C_i and C_j, respectively. The following observation will ease the situation. Suppose S is a subschedule of G containing w. Let S' be the subschedule of G obtained from S by discarding all nodes succeeding w in S. Clearly $h(S') \leq h(S)$. Therefore without loss of generality the minimum of $h(G[\Gamma] \cup \{vw\})$ can be computed over only cuts Γ such that $\Gamma(j) = w$. Moreover we can let w always be the last node of a subschedule by considering only the minimum-height j-schedule of each $G[\Gamma]$ that contains v. The first question above is no longer an issue.

It turns out that the second question is not an issue, either. We will show that in order to obtain the minimum-height of all those j-schedules, it suffices to consider only $O(\sqrt{n})$ cuts. In particular each of those $O(\sqrt{n})$ cuts is uniquely determined by its j^{th} cutpoint.

3.3 The Algorithm

The algorithm takes v and w as inputs. Let C_i contain v and C_j contain w. The algorithm proceeds iteratively with different cutpoint $\Gamma(i)$ such that $\Gamma(i)$ does not precede v. In each iteration the algorithm calls the function BEST() to obtain a minimum-height j-schedule for $G[\Gamma]$ over all cuts Γ with the designated cutpoints in C_i and C_j. By comparing the heights of these j-schedules with respect to different $\Gamma(i)$'s, the algorithm outputs the minimum height of j-schedules for $G[\Gamma]$ over all Γ such that $\Gamma(j) = w$ and $\Gamma(i)$ does not precede v. In Figure 6 we give the algorithm to compute $h(G[\Gamma^*] \cup \{vw\})$, where Γ^* is a best cut of G corresponding to vw.

Function BEST() is the essential part of the algorithm. Based on the given subset F of $\{1, 2, \ldots, p\}$ and the given cut Γ, it looks for a best cut Γ^* corresponding to vw such that $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$. (In the case that we are interested, $F = \{i, j\}$.) An optimal j-schedule of $G[\Gamma^*]$ is then returned. Note that for every $k \notin F$, $\Gamma^*(k)$ depends on a value s, which is the maximum of s_1 and s_2. Each of s_1 and s_2 is determined simply by chains with indices in F and their designated cutpoints. Namely the choices of $\Gamma^*(k)$'s for different $k \notin F$ are mutually independent. This is the key to our efficient algorithm.

In BEST(), we do not explicitly specify cutpoints of Γ^*. Instead, we work on hump representation of subchains and every cutpoint is implicitly specified by an *end*(H) for some hump H. Specifically, Step 1 implies that $\Gamma^*(k)$ is $\Gamma(k)$ for every $k \in F, k \neq j$. Step 3 and Step 8 imply that $\Gamma^*(k)$ is *end*(H), where H is the highest N-hump of C_k that has height less than s, for every $k \notin F$. Because we are considering j-schedules, $\Gamma^*(j)$ is specified slightly differently. Although in Step 2 the subchain of C_j is only up to *pred*$(\Gamma(j))$, $\Gamma^*(j)$ is still $\Gamma(j)$, since j-schedule $S^*\Gamma(j)$ is returned in Step 10.

3.4 Correctness

We answer the following two questions in this subsection:

1. Why is it sufficient to try for $\Gamma(i)$ only those nodes that are *end*(H) for $H \in I_0$?

Function $\textsc{MinHeight}(v, w)$

1 C_i := the chain containing v;
2 C_j := the chain containing w;
3 $\Gamma(j) := w$;
4 $h^* := \infty$;
5 $I_0 := \{v\} \cup \textsc{Decomp}([\mathit{succ}(v), -])$;
6 For every $\Gamma(i) \in \{\mathit{end}(H) : H \in I_0\}$ do
7 $\quad S^* := \textsc{Best}(j, \{i, j\}, \Gamma)$;
8 $\quad h^* := \min\{h^*, h(S^*)\}$;
9 Return h^*;

Function $\textsc{Best}(j, F, \Gamma)$

1 $I := \bigcup_{k \in F, k \neq j} \textsc{Decomp}([-, \Gamma(k)])$;
2 $J := \textsc{Decomp}([-, \mathit{pred}(\Gamma(j))])$;
3 $K := \bigcup_{k \notin F} \textsc{Decomp}(C_k)$;
4 $s_1 := \max\{h(H) : H \in I \cup J, c(H) < 0\}$;
5 $S^+ := \textsc{Merge}(\{H \in I \cup J : c(H) \geq 0\})$;
6 $s_2 := h(S^+\Gamma(j))$;
7 $s := \max\{s_1, s_2\}$;
8 $K_s := \{H \in K : h(H) < s, c(H) < 0\}$;
9 $S_s := \textsc{Merge}(I \cup J \cup K_s)$;
10 Return $S_s\Gamma(j)$;

Figure 6: The algorithm to compute $h(G[\Gamma^*] \cup \{vw\})$ for a best cut Γ^* of G corresponding to vw.

2. Why does $\textsc{Best}(j, F, \Gamma)$ return an optimal j-schedule of $G[\Gamma^*]$ such that $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$?

Lemma 4 Let Γ be a cut of G. Suppose $[x, z]$ is a subchain of G containing $\Gamma(i)$. Let H be the hump of $[x, z]$ containing $\Gamma(i)$. Let y be the first valley of $[\mathit{pred}(H), \Gamma(i)]$. Define Γ_1 by

$$\Gamma_1(k) = \begin{cases} \Gamma(k) & \text{if } k \neq i \\ \mathit{pred}(H) & \text{if } k = i \text{ and } y = \mathit{pred}(H) \\ \mathit{end}(H) & \text{if } k = i \text{ and } y \neq \mathit{pred}(H). \end{cases}$$

Then $h_j(G[\Gamma_1]) \leq h_j(G[\Gamma])$.

Note that the $\mathit{pred}(H)$ in the above lemma is always an $\mathit{end}(H')$ for some hump H' in I_0, which is defined in Step 5 of $\textsc{MinHeight}()$. Therefore Lemma 4 answers the first question.

By definitions of I, J, and K_s it is not difficult to see that the sequence returned by $\textsc{Best}(j, F, \Gamma)$ is an optimal j-schedule of $G[\Gamma^*]$ for some cut Γ^* such that $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$. The correctness of $\textsc{MinHeight}()$ thus relies on the following lemma, which answers the second question.

Lemma 5 Let Γ be a cut. Let F be a subset of $\{1, 2, \ldots, p\}$ containing j. Let $S^* = \textsc{Best}(j, F, \Gamma)$. Then $h(S^*) \leq h_j(G[\Gamma])$.

For the rest of the subsection, we prove Lemma 4 and Lemma 5.

We need the following lemma to prove Lemma 4.

Lemma 6 Let Γ be a cut of G. Suppose x is a node in C_i preceding $\Gamma(i)$. Define Γ_1 by

$$\Gamma_1(k) = \begin{cases} \Gamma(k) & \text{if } k \neq i \\ \text{the first valley of } [x, \Gamma(i)] & \text{if } k = i. \end{cases}$$

Then $h_j(G[\Gamma_1]) \leq h_j(G[\Gamma])$.

Proof By the second hump decomposition property, there exists a series of nodes in C_i, $\Gamma_1(i) = y_1, y_2, \ldots, y_m = \Gamma(i)$, such that every subchain $[\mathit{succ}(y_k), y_{k+1}]$ is a P-hump. Suppose S is an optimal j-schedule of $G[\Gamma]$. Let S' be obtained from S by clustering every P-hump $[\mathit{succ}(y_k), y_{k+1}]$ to its useful peak. By Lemma 1, S' is an optimal j-schedule of $G[\Gamma]$. Let S_1 be obtained from S' by removing every P-hump $[\mathit{succ}(y_k), y_{k+1}]$. Clearly $h(S_1) \leq h(S')$. Since S_1 is a j-schedule of $G[\Gamma_1]$, $h_j(G[\Gamma_1]) \leq h(S_1) \leq h(S') = h_j(G[\Gamma])$. $\square$

Lemma 4 can be proved as follows.

Proof of Lemma 4 When $y = \mathit{pred}(H)$, by choice of y, $h_j(G[\Gamma_1]) \leq h_j(G[\Gamma])$ is immediate from Lemma 6.

If H is a P-hump, by definition of $\textsc{Decomp}()$, $\mathit{pred}(H)$ is the first valley of $[\mathit{pred}(H), \mathit{end}(H)]$, so $y = \mathit{pred}(H)$. Therefore when $y \neq \mathit{pred}(H)$, H must be an N-hump. We claim that $[\mathit{start}(H), y]$ is a hump. Since y is the first valley of $[\mathit{pred}(H), \Gamma(i)]$

and $y \neq \textit{pred}(H)$, y is also the first valley of $[\textit{start}(H), \Gamma(i)]$. By definition of humps, y cannot precede any useful peak of H. It follows that a useful peak of H is also a useful peak of $[\textit{start}(H), y]$. Thus $[\textit{start}(H), y]$ is a hump. Let us use H' to denote $[\textit{start}(H), y]$. Clearly $h(H') = h(H)$. Since H is an N-hump, $c(H') \geq c(H)$.

Let us define Γ' by

$$\Gamma'(k) = \begin{cases} \Gamma(k) & \text{if } k \neq i \\ y & \text{if } k = i. \end{cases}$$

By lemma 6, $h_j(G[\Gamma']) \leq h_j(G[\Gamma])$. Suppose S is an optimal j-schedule of $G[\Gamma']$ in which H' is clustered. We write $S = S_1 H' S_2$. Inserting the sequence $[\textit{succ}(y), \textit{end}(H)]$ immediately after H', we obtain a j-schedule $S^* = S_1 H S_2$ for $G[\Gamma_1]$. We show $h(S^*) \leq h(S)$.

Now $h(S^*)$ is equal to the maximum of $h(S_1)$, $c(S_1) + h(H)$, and $c(S_1 H) + h(S_2)$. Clearly

$$h(S_1) \leq h(S). \tag{1}$$

Since $h(H) = h(H')$,

$$c(S_1) + h(H) \leq h(S). \tag{2}$$

Since $c(H) \leq c(H')$, $c(S_1 H) \leq c(S_1 H')$. Hence

$$c(S_1 H) + h(S_2) \leq h(S). \tag{3}$$

Combining (1), (2), and (3), we obtain $h(S^*) \leq h(S)$. $\square$

Let F_ℓ denote the set $\{1, \ldots, \ell-1, \ell+1, \ldots, p\}$. The following lemma is a special case of Lemma 5, in which F is composed of $p-1$ numbers.

Lemma 7 Let Γ be a cut. for some $\ell \neq j$. Let $S^* = \text{BEST}(j, F_\ell, \Gamma)$. Then $h(S^*) \leq h_j(G[\Gamma])$.

Proof Define Γ_1 by

$$\Gamma_1(k) = \begin{cases} \Gamma(k) & \text{if } k \neq \ell \\ \text{the first valley of } [-, \Gamma(\ell)] & \text{if } k = \ell. \end{cases}$$

Then by Lemma 6,

$$h_j(G[\Gamma_1]) \leq h_j(G[\Gamma]). \tag{4}$$

Let Γ^* be the cut such that S^* is a j-schedule of $G[\Gamma^*]$. Thus

$$h(S^*) = h_j(G[\Gamma^*]). \tag{5}$$

By definition of $\text{BEST}()$, Γ^* and Γ_1 could differ only at the ℓ^{th} components. We show

$$h_j(G[\Gamma^*]) \leq h_j(G[\Gamma_1]). \tag{6}$$

Let $w = \Gamma_1(j)$. Let $L = \text{DECOMP}([-, \Gamma_1(k)])$. Define

$$S = \text{MERGE}(I \cup J \cup L),$$

where I and J are defined in Step 1 and Step 2 of $\text{BEST}()$. Clearly Sw is an optimal j-schedule of $G[\Gamma_1]$. Thus $h(Sw) = h_j(G[\Gamma_1])$. By choice of $\Gamma_1(\ell)$, L contains no P-hump. Hence by the uniqueness assumption of $\text{MERGE}()$, we could write

$$Sw = S_1 S^+ w,$$

where S^+ is defined in Step 5 of $\text{BEST}()$. We prove (6) by showing the following two claims.

Claim 1 *If $\Gamma^*(\ell)$ succeeds $\Gamma(\ell)$, then $h_j(G[\Gamma^*]) \leq h(Sw)$.*

Since L contains no P-hump, each hump of $[-, \Gamma_1(\ell)]$ appears in S_1. Therefore we obtain a j-schedule of $G[\Gamma^*]$ from Sw by inserting the sequence $S' = [\textit{succ}(\Gamma_1(\ell)), \Gamma^*(\ell)]$ immediately before S^+, obtaining

$$S_1 S' S^+ w.$$

We show $h(S_1 S' S^+ w) \leq h(S_1 S^+ w)$.

Now $h(S_1 S' S^+ w)$ is equal to the maximum of $h(S_1)$, $c(S_1) + h(S')$, and $c(S_1 S') + h(S^+ w)$. Clearly

$$h(S_1) \leq h(S_1 S^+ w). \tag{7}$$

By definition of F, the K_s defined in Step 8 of $\text{BEST}()$ is composed of the N-humps of C_ℓ that have heights less than s. Therefore by choice of $\Gamma^*(\ell)$ every hump of $[-, \Gamma^*(\ell)]$ has height less than s. It follows from the standard order of humps in S' that $h(S') < s$. By Step 7 of $\text{BEST}()$, $s = \max\{s_1, s_2\}$. If $s = s_2 = h(S^+ w)$, as defined in Step 6 of $\text{BEST}()$, then $c(S_1) + h(S') < c(S_1) + h(S^+ w)$. If $s = s_1 = h(H^*)$, where H^* is a highest N-hump in $I \cup J$, then we could write $S_1 = S_2 H^* S_3$. It follows that

$$\begin{aligned} c(S_1) + h(S') &= c(S_2 H^* S_3) + h(S') \\ &< c(S_2) + h(H^*) \\ &\leq h(S_2 H^*) \\ &\leq h(S_1). \end{aligned}$$

Therefore in either case

$$c(S_1) + h(S') < h(S_1 S^+ w). \tag{8}$$

By choice of $\Gamma^*(\ell)$, $c(S') < 0$. Hence

$$\begin{aligned} c(S_1 S') + h(S^+ w) &< c(S_1) + h(S^+ w) \\ &\le h(S_1 S^+ w). \end{aligned} \tag{9}$$

Combining (7), (8), and (9), we obtain $h(S_1 S' S^+ w) \le h(Sw)$. Claim 1 is thus proved.

Claim 2 *If $\Gamma^*(\ell)$ precedes $\Gamma_1(\ell)$, then $h_j(G[\Gamma^*]) \le h(Sw)$.*

Let $S' = [succ(\Gamma^*(\ell)), \Gamma_1(\ell)]$. By choice of $\Gamma^*(\ell)$, it is not difficult to see that

$$\begin{aligned} &\text{DECOMP}([-, \Gamma_1(\ell)]) = \\ &\quad \text{DECOMP}([-, \Gamma^*(k)]) \cup \text{DECOMP}(S'). \end{aligned}$$

By choice of $\Gamma_1(\ell)$, DECOMP(S') contains only N-humps of heights no less than s. Note that every N-hump in $I \cup J$ has height no less than s. By standard form of S, S' is a suffix of S_1. Therefore we could write $Sw = S_2 S' S^+ w$. Removing S' from Sw, we obtain a j-schedule $S_2 S^+ w$ of $G[\Gamma^*]$. We show $h(S_2 S^+ w) \le h(Sw)$.

Now $h(S_2 S^+ w)$ is equal to the maximum of $h(S_2)$ and $c(S_2) + h(S^+ w)$. Clearly

$$h(S_2) \le h(S_2 S' S^+ w) = h(Sw). \tag{10}$$

Since each hump of S' has height no less than s, $h(S') \ge s$. Hence

$$h(S' S^+ w) \ge h(S') \ge s \ge s_2 = h(S^+ w).$$

It follows that

$$\begin{aligned} c(S_1) + h(S^+ w) &\le c(S_1) + h(S' S^+ w) \\ &\le h(Sw). \end{aligned} \tag{11}$$

Combining (10) and (11), we obtain $h(S_1 S^+ w) \le h(Sw)$. Claim 2 is thus proved.

Therefore (6) is followed from the above two claims. Combining (4), (5) and (6), we prove the lemma. □

Now we are ready to prove Lemma 5.

Proof of Lemma 5 Recall that S^* is equal to BEST(j, F, Γ). Suppose Γ^* is the cut such that S^* is an optimal j-schedule of $G[\Gamma^*]$. We use the algorithm in Figure 7 to prove the lemma. Procedure CUTTRANS() proceeds with iterations, in which the value of ℓ varies among $\{1, \ldots, p\}$. If $\ell \notin F$, then the value of $\Gamma(\ell)$ is updated. Since S is an optimal j-schedule of $G[\Gamma']$, it follows from Lemma 7 that $h_j(G[\Gamma']) \le h_j(G[\Gamma])$ always holds during the while-loop. If we could show that CUTTRANS() always terminates, then the lemma is proved.

Let s_1^*, s_2^*, and s^* be the s_1, s_2, and s in the execution of BEST(j, F, Γ). Let s_1, s_2, and s be those in the execution of BEST(j, F_ℓ, Γ). The values of s_1, s_2, and s change as the while-loop of CUTTRANS() proceeds. We show Γ eventually becomes Γ^* by arguing that s eventually becomes s^*.

Since $F \subseteq F_\ell$, $s_1 \ge s_1^*$ always holds. By definition of BEST(), whenever Step 7 of CUTTRANS() is finished, $[-, \Gamma(\ell)]$ contains only N-humps. Thus after the first p iterations of the while-loop, $[-, \Gamma(\ell)]$ contains no P-hump for every $\ell \notin F$. Henceforth $s_2 = s_2^*$ and therefore $s = \max\{s_1, s_2\} \ge \max\{s_1^*, s_2^*\} = s^*$. If $s > s^*$, then $s = s_1 > s^*$. Since $s_1 > s^*$, there must be an N-hump H in $\bigcup_{k \notin F}$ DECOMP$([-, \Gamma(k)])$ such that $h(H) = s_1$. Since $s = s_1$, in the next iteration when C_ℓ contains H, $\Gamma(\ell)$ will be moved before H by definition of BEST(). It follows that the value of s is nonincreasing and s will become s^*. Once $s = s^*$, in the following p iterations, $\Gamma(k)$ will be moved to $\Gamma^*(k)$ for every $k \notin F$. The algorithm then terminates. □

3.5 Implementation

Recall that DECOMP(C) runs in time linear in $|C|$, the length of chain C. It follows that the time complexity of Step 1–5 and Step 9 of MINHEIGHT() is $O(n)$. Suppose the order of nodes assigned to $\Gamma(i)$ in the for-loop is the same as their order in C_i. In the subsection we focus on implementing BEST() such that the for-loop runs in time $O(n)$.

Number of Iterations The following lemma ensures that the size of I_0 is $O(\sqrt{|C_i|})$. It follows that the number of iterations is $O(\sqrt{n})$.

```
Procedure CUTTRANS(Γ, Γ*)
ℓ := 0;
While Γ* ≠ Γ do
    ℓ := (ℓ mod p) + 1;
    If ℓ ∉ F
        S := BEST(j, F_ℓ, Γ);
        Γ' := the cut such that S is an
            optimal j-schedule of G[Γ'];
        Γ := Γ';
```

Figure 7: The algorithm transforms Γ to Γ^*. We prove Lemma 5 by showing that this algorithm always terminates.

Lemma 8 Suppose C is a chain with node costs ± 1. The number of humps in DECOMP(C) is $O(\sqrt{|C|})$.

Proof Since the costs of nodes are either $+1$ or -1, a hump of height ℓ contains at least ℓ nodes. For the same reason a hump of reverse height ℓ contains at least ℓ nodes. By the first hump decomposition property, the heights of the N-humps in DECOMP(C) are different, and so are the reverse heights of the P-humps in DECOMP(C). If there are n_1 N-humps and n_2 P-humps in DECOMP(C), then $|C| = \Omega(n_1^2 + n_2^2) = \Theta((n_1+n_2)^2)$. This proves the lemma. □

Compact Representation of Humps For the sake of efficiency we do not deal with the internal structure of humps in BEST(). It suffices to represent each hump H by a pair $(c(H), h(H))$ and work on the compact representation of humps. Therefore each of the I, J, and K computed in the first three steps is a set of pairs. Clearly each of these three steps takes $O(n)$ time. However, the contents of J and K do not change in different iterations. Thus Step 2 and Step 3 need only be executed once.

Since $F = \{i, j\}$, $I =$ DECOMP$([-, \Gamma(i)])$. Suppose I_t and Γ_t are the I and Γ in the t^{th} iteration for some $t \geq 2$. By the order of nodes assigned to $\Gamma(i)$, we need not recompute DECOMP$([-, \Gamma_t(i)])$ from scratch. In the t^{th} execution of Step 1, $[-, \Gamma_t(i)]$ is obtained by appending a hump $[succ(\Gamma_{t-1}(i)), \Gamma_t(i)]$ to $[-, \Gamma_{t-1}(i)]$. By the argument following the hump decomposition properties in Subsection 2.2 the t^{th} execution of Step 1 takes $O(|I_{t-1}|)$ time. It follows from Lemma 8 that the time complexity of all executions of Step 1 is $O(n + \sqrt{n} \times \sqrt{n}) = O(n)$.

Priority Tree To compute s_1 efficiently, we need a *priority tree*, a complete binary tree with $n+1$ leaves. Each leaf keeps two values, *count* and *maxheight*. The cost of $h+1^{st}$ leaf is the number of N-humps of height h in $I \cup J$. The maxheight of the $h+1^{st}$ leaf is 0 and h, if its count is zero and nonzero, respectively. The maxheight of an internal node is the maximum maxheight of its children. It follows that the maxheight of the root of priority tree is the correct value of s. The priority tree can be built in time $O(n)$. Whenever a hump is added to or deleted from $I \cup J$, the priority tree can be updated in time $O(\log n)$. Since J is fixed, to compute s_1 in t^{th} iteration for every $t \geq 2$, we add humps in $I_t - I_{t-1}$ to $I \cup J$, remove humps in $I_{t-1} - I_t$ from $I \cup J$, and update the priority tree. By the third hump decomposition property

$$\sum_{2 \leq t \leq p} |I_t - I_{t-1}| + |I_{t-1} - I_t| = O(\sqrt{|C_i|}). \quad (12)$$

Hence the time complexity of all executions of Step 4 is $O(n + \sqrt{n} \times \log n) = O(n)$.

Hump Tree To obtain the value of s_1, it is not necessary to know the value of S^+. We need only to obtain the height of $S^+\Gamma(j)$. Similarly the actual value of S_s is irrelevant. What we compare in Step 8 of MINHEIGHT() is the height of $S_s\Gamma(j)$. We need a data structure to compute these two heights efficiently.

Let L be a set of humps such that $h(H) \leq n$ and $\tilde{h}(H) \leq n$ for every $H \in L$. A *hump tree* T for L is a binary tree composed of two complete binary subtrees. Each subtree has $n+1$ leaves. Let T_N be the left subtree and T_P be the right subtree. The $h+1^{st}$ leaf of T_N associates with the set of N-humps of height h in L. The $h+1^{st}$ leaf of T_P associates with the set of P-humps of reverse height $n-h$ in L. Let T_x be the subtree of T rooted at x. Let L_x be the set of humps associated with leaves of T_x. Define $h(T_x) = h($MERGE$(L_x))$ and $c(T_x) = c($MERGE$(L_x))$. Clearly when $L = I \cup J$,

$h(T_P) = h(S^+)$ and $c(T_P) = c(S^+)$. When $L = I \cup J \cup K_s$, $h(T) = h(S_s)$ and $c(T) = c(S_s)$. The heights of $S^+\Gamma(j)$ and $S_s\Gamma(j)$ can then be computed by

$$\begin{aligned} h(S^+\Gamma(j)) &= \max\{h(S^+), c(S^+) + h(\Gamma(j))\} \\ h(S_s\Gamma(j)) &= \max\{h(S_s), c(S_s) + h(\Gamma(j))\}. \end{aligned}$$

Let us keep $h(T_x)$ and $c(T_x)$ in x for every node x of T. Therefore the hump tree T takes $O(n)$ space. We show how to compute $h(T_x)$ and $c(T_x)$ for every node x from leaves to root. When x is a leaf of T, the humps in L_x have the same height if x is in T_N, and the same reverse height if x is in T_P. It is not difficult to see that $c(T_x) = \sum_{H \in L_x} c(H)$; and $h(T_x)$ is equal to

$$\begin{cases} 0 & \text{if } L_x = \emptyset \\ h & \text{if } x \text{ is the } h + 1^{st} \text{ leaf of } T_N \\ c(T_x) - h & \text{if } x \text{ is the } n - h + 1^{st} \text{ leaf of } T_P. \end{cases}$$

When x is an internal node of T, $h(T_x)$ and $c(T_x)$ can be computed by the information kept in the children of x. Suppose y and z are the left and right children of x, respectively. For any H in L_y and H' in L_z, by the way we associate humps with leaves, the series HH' is in standard order. Hence

$$\begin{aligned} h(T_x) &= \max\{h(T_y), c(T_y) + h(T_z)\} \\ c(T_x) &= c(T_y) + c(T_z). \end{aligned}$$

It follows that the hump tree T for L can be built in time $O(n + |L|)$.

Once T is built, inserting a hump to L can be done efficiently. Suppose we insert H to L. If H is an N-hump then we add $c(H)$ to $c(T_x)$ where x is the $h(H) + 1^{st}$ leaf of T_N. If H is a P-hump then we add $c(H)$ to both $c(T_x)$ and $h(T_x)$, where x is the $n - \tilde{h}(H) + 1^{st}$ leaf of T_P. To update T, we simply update the internal nodes on the path from x to the root of T. Deleting a hump from L can be done similarly by replacing every addition with a subtraction. Clearly both insertion and deletion take time $O(\log n)$.

To compute the heights of $S^+\Gamma(j)$ and $S_s\Gamma(j)$, we need not maintain a hump tree for $I \cup J$ and another hump tree for $I \cup J \cup K_s$. Suppose K^- is the set of N-humps in K, i.e. $K^- = \{H \in K : c(H) < 0\}$. It suffices to maintain a hump tree T for $I \cup J \cup K^-$.

```
Procedure RemoveRange(T, s)
y := the s^th leaf of T_N;
While y is not the root of T_N do
    x := the parent of y;
    If y is the left child of x then
        (h(T_x), c(T_x)) := (h(T_y), c(T_y));
    else
        Recompute h(T_x) and c(T_x);
    y := x;
Recompute h(T) and c(T);
```

Figure 8: Let T be the hump tree for $I \cup J \cup K^-$. This procedure acts as if the N-humps of heights no less than s are removed from the hump tree.

Since there is no P-hump in K^-, it is still true that $h(T_p) = h(S^+)$ and $c(T_p) = c(S^+)$. Although the hump tree is not for $I \cup J \cup K_s$, the values of $h(S_s)$ and $c(S_s)$ can be efficiently obtained by the procedure in Figure 8. Procedure RemoveRange() acts as if the N-humps of heights no less than s are removed from the hump tree for $I \cup J \cup K^-$. Therefore the resulting $h(T)$ and $c(T)$ are $h(S_s)$ and $c(S_s)$, respectively. Clearly RemoveRange() takes $O(\log n)$ time. Since we maintain the hump tree for $I \cup J \cup K^-$ in every iteration, we use $O(\log n)$ space to keep those modified information of T. After obtaining the information we need, we restore the hump tree for $I \cup J \cup K^-$ in time $O(\log n)$.

Let I_t be the I in the t^{th} iteration for any $t \geq 1$. To obtain the hump tree for $I_t \cup J \cup K^-$ from $I_{t-1} \cup J \cup K^-$, we need to insert the humps in $I_t - I_{t-1}$ to T and remove the humps in $I_{t-1} - I_t$ from T. Since each insertion and deletion takes $O(\log n)$ time, it follows from (12) that the overall time complexity for obtaining the hump tree from that of previous iteration is $O(\sqrt{n} \times \log n)$. Recall that building a hump tree for L takes $O(n + |L|)$ time. Since there are n nodes in G, $|I_1 \cup J \cup K^-| = O(n)$. It follows that the time complexity for building a hump tree for $I_1 \cup J \cup K^-$ is $O(n)$.

By the above arguments we implement Best() such that the overall time complexity of the while-loop in MinHeight() is $O(n)$. Theorem 2 is thus proved.

4 Algorithm for All Pairs

Recall that G is composed of p disjoint chains, $C_1, C_2, \ldots, C_p$, of n nodes. In this section we show how to determine the $\to$ relations for all pairs of nodes in G. The linear-time algorithm for a single pair of nodes, applied to all $O(n^2)$ pairs, takes time $O(n^3)$. We present a faster algorithm, one that runs in time $O(np \log n)$.

4.1 The Algorithm

Let $\mathit{first}_j(v)$ denote the first node in C_j that could be preceded by v in some valid subschedule of G. The output of the all-pair algorithm is thus the value of $\mathit{first}_j(v)$ for every node v and $1 \le j \le p$. Note that $\mathit{first}_j(v)$ could be $\top$, which means that none of nodes in C_j can be preceded by v in any valid subschedule of G.

Let us describe first the procedure ChainPair(i, j) which computes $\mathit{first}_j(v)$ for every $v \in C_i$. The all-pair algorithm is simply calling ChainPair(i, j) for every $1 \le i, j \le p$. For convenience, let $\mathit{succ}_j(w) = \mathit{succ}(w)$ for every $w \in C_j$ and let $\mathit{succ}_j(\perp) = \mathit{start}(C_j)$. Procedure ChainPair$(i, j)$ is shown in Figure 9. The algorithm starts with letting v be $\mathit{end}(C_i)$ and letting w be $\mathit{end}(C_j)$. The repeat-loop proceeds by replacing w with $\mathit{pred}(w)$. Once MinHeight(v, w) is not zero, the algorithm reports $\mathit{succ}_j(w)$ as $\mathit{first}_j(v)$. After replacing v with $\mathit{pred}(v)$, the repeat-loop continues the same procedure to search for new $\mathit{first}_j(w)$.

```
Procedure ChainPair(i, j)
   (v, w) := (end(C_i), end(C_j));
   Repeat
      If w = ⊥ then h := 1;
                else h := MinHeight(v, w);

      If h > 0  then first_j(v) := succ_j(w);
                     v := pred(v);
                else w := pred(w);
   Until v = ⊥;
```

Figure 9: The algorithm to compute $\mathit{first}_j(v)$ for every $v \in C_i$

4.2 Correctness

By induction on v we show that ChainPair(i, j) correctly computes $\mathit{first}_j(v)$ for every $v \in C_i$.

When $v = \mathit{end}(C_i)$, procedure ChainPair(i, j) keeps replacing w with $\mathit{pred}_j(w)$ until $w = \perp$ or MinHeight$(v, w) > 0$. If $w = \perp$ then $h(G \cup \{vw'\}) = 0$ for every $w' \in C_j$. Thus $\mathit{first}_j(v) = \mathit{succ}_j(\perp) = \mathit{succ}_j(w) = \mathit{start}(C_j)$ is correct. If MinHeight$(v, w) > 0$ then $v \not\to w$. It follows that $v \not\to w'$ for every w' precedes w in C_j. Since MinHeight$(v, \mathit{succ}_j(w)) = 0$, $v \to \mathit{succ}_j(w)$. Therefore $\mathit{succ}_j(w)$ is the correct value of $\mathit{first}_j(v)$. This confirms the induction basis.

Suppose the procedure ChainPair(i, j) correctly reports $\mathit{succ}_j(w)$ as the value of $\mathit{first}_j(\mathit{succ}_i(v))$ in a certain iteration of the repeat-loop. We need to show that in the remaining iterations $\mathit{first}_j(v)$ will also be correctly computed. Since $\mathit{succ}_i(v) \to \mathit{succ}_j(w)$, $v \to \mathit{succ}_j(w)$. It follows that $v \to w'$ (and thus MinHeight$(v, w') = 0$) for every w' succeeding w in C_j. In other words, to locate the first node in C_j that could be preceded by v, it suffices to start testing from w. For the same reason as above, ChainPair(i, j) reports the correct value of $\mathit{first}_j(w)$. The correctness is therefore ensured.

4.3 Implementation

We show in this subsection how to implement ChainPair(i, j) to run in time $O((|C_i|+|C_j|)\log n)$. It then follows that the time complexity of the all-pair algorithm is $O(np \log n)$.

Suppose each time before we call ChainPair(i, j), we have the hump tree for $I \cup J \cup K^-$, where

$$\begin{aligned} I &= \textsc{Decomp}(C_i) \\ J &= \textsc{Decomp}([-, \mathit{pred}(\mathit{end}(C_j))]) \\ K^- &= \{H \in \bigcup_{\substack{1 \le k \le p \\ k \ne i,j}} \textsc{Decomp}(C_k) : c(H) < 0\}. \end{aligned}$$

It follows from Subsection 3.5 that the first call to MinHeight(v, w) can be computed in time $O(\log n)$, since only one $\Gamma(i)$ need be considered. In each of the remaining iterations of the repeat-loop, we either replace v with $\mathit{pred}(v)$ or replace w

with $pred(w)$. The remaining lemma guarantees that to compute each of the following $\text{MINHEIGHT}(v, w)$, we need only try v as the cutpoint of C_i.

Lemma 9 Consider any iteration of the repeat-loop in $\text{CHAINPAIR}(i, j)$. When the algorithm computes $h = \text{MINHEIGHT}(v, w)$, v is the only cutpoint of C_i that could make h zero.

Proof By definition of $\text{CHAINPAIR}()$, when computing $\text{MINHEIGHT}(v, w)$, $\textit{first}_j(\textit{succ}_i(v))$ always succeeds w in C_j. Assume for a contradiction that u is a node succeeding v in C_i such that there is a cut Γ of G where $\Gamma(i) = u$, $\Gamma(j) = w$, and $h_j(G[\Gamma]) = 0$. It follows that $u \to w$ and thus $\textit{succ}_i(v) \to w$. This contradicts the fact that $\textit{first}_j(\textit{succ}_i(v))$ succeeds w in C_j. □

Note that in each iteration of the repeat-loop, either v or w is moved by one position. Since the costs of v and w are ± 1, by the first hump decomposition property the number of humps updated in $I \cup J \cup K^-$ between two consecutive iterations is a constant. Thus each execution of $\text{MINHEIGHT}(v, w)$ takes only time $O(\log n)$. Since the number of iterations of the repeat-loop is $O(|C_i| + |C_j|)$, each execution of $\text{CHAINPAIR}(i, j)$ takes time

$$O((|C_i| + |C_j|) \times \log n). \tag{13}$$

The remaining thing is how to efficiently build the hump tree for each execution of $\text{CHAINPAIR}(i, j)$.

The very first hump tree can be constructed in time

$$O(n). \tag{14}$$

Consider the moment when $\text{CHAINPAIR}(i, j)$ is just finished and the all-pair algorithm is about to call $\text{CHAINPAIR}(i_1, j_1)$. The current T is the hump tree for

$$\{H \in \bigcup_{\substack{1 \le k \le p \\ k \ne i,j}} \text{DECOMP}(C_k) : c(H) < 0\},$$

since all humps in $I \cup J$ have been deleted during the execution of $\text{CHAINPAIR}(i, j)$. To obtain the hump tree for $\text{CHAINPAIR}(i_1, j_1)$, we have to add the humps in

$$\{H \in \text{DECOMP}(C_i) \cup \text{DECOMP}(C_j) : c(H) < 0\}$$

to T, delete the humps in

$$\{H \in \text{DECOMP}(C_{j_1}) : c(H) < 0\}$$

from T, and then insert the humps in

$$\{H \in \text{DECOMP}(C_{i_1}) : c(H) \ge 0\} \cup \text{DECOMP}([-, \textit{pred}(\textit{end}(C_{j_1}))])$$

to T. The hump decomposition can be done in time

$$O(|C_i| + |C_j| + |C_{i_1}| + |C_{j_1}|). \tag{15}$$

The insertion and deletion of humps can be done in time

$$O((\sqrt{|C_i|} + \sqrt{|C_j|} + \sqrt{|C_{i_1}|} + \sqrt{|C_{j_1}|}) \times \log n). \tag{16}$$

Combining (13), (14), (15), and (16), we obtain that the overall time complexity of the all-pair algorithm is

$$O(n) + \sum_{1 \le i,j \le p} O(|C_i| + |C_j|) + \sum_{1 \le i,j \le p} O(\sqrt{|C_i|} + \sqrt{|C_j|}) \times \log n + \sum_{1 \le i,j \le p} O(|C_i| + |C_j|) \times \log n,$$

which is $O(np \log n)$. Theorem 3 is proved.

5 NP-completeness

In this section we show that determining whether $v \to w$ for chain graphs of operations on more than one semaphore. The proof is by reduction from the NP-complete uniform-cost SMMCC problem, where the node costs are restricted to ± 1 [5]. The reduction has three steps. Given a SMMCC problem for a uniform-cost graph G_0 of n nodes, we construct $O(\log n)$ chain graphs with $n + 2$ semaphores. The first step of the reduction shows that the SMMCC problem for G_0 can be reduced to determining whether each of those $O(\log n)$ chain graphs has a valid schedule. The second step shows that each of those $O(\log n)$ chain graphs can be *simulated* by a chain graph with only two semaphores. In other words, the simulated chain graph has a valid schedule if and only if the simulating chain graph has a valid schedule. The last step shows that the simulating chain graph has a valid schedule if and only if $v \to w$, for some v and w, in the same chain graph. We will show the same proof works even if the $\to$ is defined for schedule as in [8]

5.1 Definition and Notation

Let G be a chain graph. Each node of G is an operation on a semaphore. An operation on semaphore S is either $+S$, incrementing the value of S by one, or $-S$, decrementing the value of S by one. A subschedule of G is *valid* if the value of each semaphore is always nonpositive during the execution of the subschedule. Let v and w be two nodes of G. If there exists a partial schedule of G in which v precedes w, then we say $v \to w$. Clearly determining whether $v \to w$ is in NP. If G is allowed to use more than one semaphore, then we prove the NP-hardness by a three-step reduction from the uniform-cost SMMCC problem.

5.2 First Step

Let G_0 be an acyclic directed graph of n nodes, $v_1, v_2, \ldots, v_n$. The cost of each node is either $+1$ or -1. Suppose we would like to know whether $h(G_0) \leq \ell$. We construct a chain graph G_1 composed of $2n+2$ chains of operations on $n+2$ semaphores, and argue that G_1 has a valid schedule if and only if $h(G_0) \leq \ell$. Note that $0 \leq h(G_0) \leq n$. Therefore $h(G_0)$ can be obtained by $O(\log n)$ queries of whether a chain graph of $n+2$ semaphores has a valid schedule.

Let n^+ be the number of nodes with positive costs. Let n^- be the number of nodes with negative costs. Clearly $n^+ - n^-$ is the sum of node costs of G_0. Let d_i be the number of outgoing arcs of G_0 from v_i. The $n+2$ semaphores for G_1 are $S_1, S_2, \ldots, S_{n+2}$. To distinguish the last two semaphores, we also write $S_\alpha = S_{n+1}$ and $S_\beta = S_{n+2}$. Let the $2n+2$ chains of G_1 be $C_1, \ldots, C_{n+1}$, and $C'_1, \ldots, C'_{n+1}$, all initially empty. We construct G_1 from G_0 by the procedure CONSTRUCT() in Figure 10, which runs in polynomial time. Without loss of generality we can assume that $\ell - n^+ + n^-$, the number in the second-to-last statement of the procedure CONSTRUCT, is nonnegative, since otherwise $h(G_0) > \ell$.

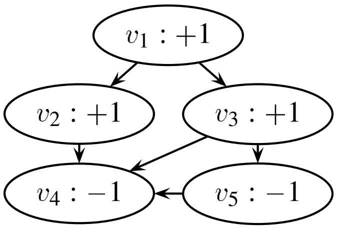
```
CONSTRUCT(G_0)
  For i := 1 to n do
    For j := 1 to n do
      If v_j v_i is an arc of G_0 then
        Append a +S_j to C_i.
    If the cost of v_i is +1 then
      Append a +S_α to C_i.
    else (i.e. the cost of v_i is −1)
      Append a −S_α to C_i.
      Append a +S_α and −S_α to C'_i.
    Append d_i copies of −S_i to C_i.
    Append a −S_β to C_i.
  Append n copies of +S_β to C_{n+1}.
  Append ℓ − n^+ + n^− copies of +S_α to C_{n+1}.
  Append ℓ copies of −S_α to C'_{n+1}.
```

Figure 10: The procedure constructs a chain graph G_1 such that G_1 has a valid schedule if and only if $h(G_0) \leq \ell$.

An example is shown in Figure 11. The intuition is as follows. The (only) operation for S_α in C_i corresponds to v_i, where the "sign" of S_α reflects the cost of v_i. We use the first n semaphores, $S_1, \ldots, S_n$, to enforce the execution of these n operations for S_α to obey the precedence constraints imposed by G_0. In Figure 11, for instance, in order to reach the $-S_\alpha$ in C_4, we have to unlock the $+S_2$ (and $+S_3$, $+S_5$) in the same chain first. Since the only $-S_2$ is after the $+S_\alpha$ in C_2, we know the $+S_\alpha$ in C_2 must be executed before the $-S_\alpha$ in C_4.

The $-S_\beta$'s at the end of $C_1, \ldots, C_n$ are to ensure that as long as the last $+S_\beta$ in C_{n+1} is executed, all operations in $C_1, \ldots, C_n$ are already executed. The function of those ℓ copies of $-S_\alpha$ in C'_{n+1} is clear: The larger ℓ, the easier for G_1 to have a valid schedule. The purpose of the $+S_\alpha, -S_\alpha$ pairs in $C'_1, \ldots, C'_n$ and those $\ell - n^+ + n^-$ copies of $+S_\alpha$'s at the end of C_{n+1} will become clear as we proceed. Basically they are used to ensure that G_1 has some kind of "pairwise" schedule, as long as G_1 has a valid schedule. One can verify that there are the same number of $+S_i$'s and $-S_i$'s in G_1, for each $1 \leq i \leq n+2$.

For the rest of the subsection we prove that



C_1	C_2	C_3	C_4	C_5	C_6	C'_4	C'_5	C'_6
			$+S_2$		$+S_\beta$	$+S_\alpha$	$+S_\alpha$	$-S_\alpha$
			$+S_3$		$+S_\beta$	$-S_\alpha$	$-S_\alpha$	$-S_\alpha$
	$+S_1$	$+S_1$	$+S_5$	$+S_3$	$+S_\beta$			
					$+S_\beta$			
$+S_\alpha$	$+S_\alpha$	$+S_\alpha$	$-S_\alpha$	$-S_\alpha$	$+S_\beta$			
$-S_1$	$-S_2$	$-S_3$		$-S_5$				
$-S_1$		$-S_3$						
$-S_\beta$	$-S_\beta$	$-S_\beta$	$-S_\beta$	$-S_\beta$	$+S_\alpha$			

Figure 11: An example for the first step of the reduction. Suppose we would like to determine whether $h(G_0) \leq 2$, where G_0 is the graph at left. We then construct, by CONSTRUCT, the chain graph G_1 at right. Note that there are one $+S_\alpha$ at the end of C_6 and two $-S_\alpha$ in C'_6, as by the last two statements of CONSTRUCT. It follows from Lemma 10 and Lemma 12 that that exists a valid schedule of the chains at right if and only if the height of the graph at left is at most two.

$h(G_0) \leq \ell$ if and only if G_1 has a valid schedule. An implication of the following proofs is that G_1 has a valid schedule if and only if it has a valid schedule executable by some procedure PAIRWISE, which will be given in the proofs.

Lemma 10 If G_1 has a valid subschedule which contains the last $+S_\alpha$ of C_{n+1}, then $h(G_0) \leq \ell$.

Proof Let X be a valid subschedule of G_1 as described in the lemma, we show that $h(G_0) \leq \ell$. Let O_i be the operation of S_α in C_i. Since X is valid and contains the last $+S_\alpha$ of C_{n+1}, X must contain all the operations in $C_1, \ldots, C_n$. Therefore every O_i, $1 \leq i \leq n$, is in X.

Suppose the order of those O_i's in X is $O_{k_1}, O_{k_2}, \ldots, O_{k_n}$. By the definition of CONSTRUCT, if v_j is reachable from v_i in G_0, then O_j does not precede O_i in X. It follows that the sequence $Y = v_{k_1} v_{k_2} \cdots v_{k_n}$ is a schedule of G_0. Therefore it suffices to show that $h(Y) \leq \ell$.

Assume for a contradiction that $h(Y)$ is more than ℓ. If we count only those O_i's as the operations for S_α in X, the maximum value of S_α would be greater than ℓ during the execution of X. Note that there are $\ell + n^-$ other $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$, which are the only hope for bringing the maximum value of S_α down to zero. By the construction of $C'_1, \ldots, C'_n$, however, we know n^- of those $-S_\alpha$'s have to be preceded in X by n other $+S_\alpha$'s. It follows that even if we count all operations for S_α together, the maximum value of S_α would be greater than zero during the execution of X. This contradicts the fact that X is a valid schedule of G_1. □

The following lemma is immediate from Lemma 10.

Lemma 11 If G_1 has a valid schedule, $h(G_0) \leq \ell$.

Lemma 12 If $h(G_0) \leq \ell$, G_1 has a valid schedule.

Proof Let $Y = v_{k_1} v_{k_2} \cdots v_{k_n}$ be a schedule of G_0 such that $h(Y) \leq \ell$. Let m_i be the sum of costs of $v_{k_1}, \ldots, v_{k_i}$. Clearly $m_n = n^+ - n^-$, which is the sum of node costs of G_0. Since $h(Y) \leq \ell$, we know that $m_i \leq \ell$ for every $1 \leq i \leq n$. We claim that G_1 can be executed by the procedure PAIRWISE in Figure 12.

Note that in the schedule of G_1 executed by PAIRWISE, each operation $-S_i$ is immediately followed by an operation $+S_i$. Not every chain graph has such a "pairwise" schedule, however, we show that G_1 does. We first show that the first for-loop of PAIRWISE can be finished for G_1.

Procedure PAIRWISE
  For $k := k_1, k_2, \ldots, k_n$ do
    For $j := 1$ to n do
      If $v_j v_k$ is an arc of G_0 then
        Execute a $-S_k$ in C_j.
        Execute the $+S_k$ in C_k.
    If $O_k = +S_\alpha$ then
      Execute one of the $-S_\alpha$'s
        in $C'_1, C'_2, \ldots, C'_{n+1}$.
      Execute the $+S_\alpha$ in C_k.
    else (i.e. $O_k = -S_\alpha$)
      Execute the $-S_\alpha$ in C_k.
      Execute the $+S_\alpha$ in C'_k.
  For $i := 1$ to n do
    Execute the $-S_\beta$ in C_i.
    Execute a $+S_\beta$ in C_{n+1}.
  For $i := 1$ to $\ell - m_n$ do
    Execute a $-S_\alpha$ in $C'_1, \ldots, C'_{n+1}$.
    Execute a $+S_\alpha$ in C_{n+1}.

Figure 12: Procedure PAIRWISE

Lemma 13 For each $1 \le i \le n$, the i^{th} iteration of the first for-loop of PAIRWISE can be executed for G_1. Furthermore, after executing the i^{th} iteration,

- the remaining operations in C_{k_i} are d_{k_i} copies of $-S_{k_i}$'s followed by a $+S_\beta$; and
- there are $\ell - m_i$ copies of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$.

It is then not hard to see that after the execution of the first for-loop of PAIRWISE, the remaining operation in each C_i is a $-S_\beta$. Therefore the second for-loop of PAIRWISE can be finished, since there are n copies of $+S_\beta$'s available in C_{n+1}.

By Lemma 13 we know that after executing the first For-loop, the number of $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_n$, which is equal to the number of $+S_\alpha$'s at the end of C_{n+1}. Therefore the last for-loop of PAIRWISE can be finished. The lemma is proved. $\square$

It remains to prove Lemma 13

Proof of Lemma 13 We prove the lemma by induction on i. For convenience we abbreviate k_i to k for the entire proof.

When $i = 1$, we know v_k does not have any incoming arcs from other nodes. Therefore the for-loop with index j in the first iteration does not execute any operation. We then consider the if-statement.

- If $O_k = -S_\alpha$, then $c(v_k) = -1$, and thus $m_1 = -1$. There is a $+S_\alpha$ in C'_k by the definition of CONSTRUCT. We can execute the else-part of the if-statement without problem. Since the second operation in C'_k is a $-S_\alpha$, these two steps increase the number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ by one.
- If $O_k = +S_\alpha$, then $c(v_k) = 1$, and thus $m_1 = 1$. Since v_k is the first node in Y, $h(Y)$ is at least one, and thus $\ell \ge 1$. We can therefore execute the then-part of the if-statement without problem. The number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ is decreased by one.

Clearly after executing the first iteration, in which the only executed operation in C_k is O_k, the remaining operations in C_k is exactly as that described in the lemma. Note that before executing the first iteration, the number of available $-S_\alpha$'s is ℓ by the definition of CONSTRUCT. Therefore after executing the first iteration, the number of available $-S_\alpha$'s is exactly $\ell - m_1$. This confirms the inductive basis.

Let i' be an integer such that $1 < i' \le n$. Assume that the lemma holds for every $1 \le i < i'$. We show it holds for $i = i'$. Consider the i^{th} iteration. Note that for every j such that $v_j v_k$ is an arc of G_0, O_j must have been executed. By the inductive hypothesis we know those d_j copies of $-S_j$'s is already available before executing the i^{th} iteration. Therefore the for-loop with index j will proceed without problem, since there are exactly d_j copies of $+S_j$'s in G_1 by the definition of CONSTRUCT. We then consider the if-statement.

- If $O_k = -S_\alpha$, $m_i = m_{i-1} - 1$. We know there is a $+S_\alpha$ in C'_k. Thus the else-part can proceed without problem. Since the second operation in C'_k is $-S_\alpha$, these two steps increase the number of available $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ by one.

- If $O_k = +S_\alpha$, $m_i = m_{i-1} + 1$. The inductive hypothesis says that the number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_{i-1}$ before executing the i^{th} iteration. That number is at least one since $\ell - m_{i-1} - 1 = \ell - m_i \geq 0$. Therefore the then-part of the if-statement can be executed without problem. The number of available $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ is decreased by one.

Therefore the i^{th} iteration can be executed, and thus the remaining operations in C_k is as required.

It follows from the inductive hypothesis that the number of available $-S_\alpha$ in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_{i-1}$. By the above case analysis we see that the number is exactly $\ell - m_i$ after executing the i^{th} iteration. The lemma is proved. □

If G_1 has a valid schedule, by Lemma 11 we know $h(G_0) \leq \ell$. It then follows from the proof of Lemma 12 that G_1 has a valid schedule executable by PAIRWISE. Therefore we have the following lemma.

Lemma 14 G_1 has a valid schedule if and only if G_1 has a valid schedule executable by PAIRWISE.

5.3 Second Step

In this subsection we show that the G_1 constructed in the first step can be simulated by another chain graph G_2, which uses only two semaphores, T_1 and T_2. G_2 has $2n + 3$ chains. The first chain, denoted C_0, is composed of two $-T_1$'s and two $-T_2$'s. The remaining $2n + 2$ chains are obtained from those of G_1 as follows. We replace every operation $-S_i$ (and $+S_i$) by a *unit* $-U_i$ (and $+U_i$) for each $1 \leq i \leq n + 2$. Each unit, $-U_i$ or $+U_i$, is a sequence of operations on T_1 and T_2, as shown in Figure 13. We also denote those $2n + 2$ chains of G_2 by $C_1, \ldots, C_{n+1}$ and $C'_1, \ldots, C'_{n+1}$. Clearly G_2 can be constructed in polynomial time.

Note that the sequence of operations in each unit is arranged such that only a $-U_i$ and a $+U_i$ can "unlock" each other. To be more specific, suppose each of T_1 and T_2 has initial value -2, which will be the case if the four operations in C_0 are executed. Consider a graph U_{ij} for some $1 \leq i, j \leq n + 2$ composed of two units, $-U_i$ and $+U_j$, each forms a single chain. One can easily verify that U_{ij} has a valid schedule if $i = j$. (In fact it also holds for

$$
\begin{array}{l}
+T_1 \\
+T_2 \\
\left.\begin{array}{l}
\;-T_1 \\
\;+T_2 \\
\quad\vdots \\
\;-T_1 \\
\;+T_2
\end{array}\right\} i+1 \text{ pairs} \\
-T_2 \\
-T_2 \\
-T_2 \\
-T_2
\end{array}
\qquad\qquad
\begin{array}{l}
+T_1 \\
+T_2 \\
\left.\begin{array}{l}
\;+T_1 \\
\;-T_2 \\
\quad\vdots \\
\;+T_1 \\
\;-T_2
\end{array}\right\} i+1 \text{ pairs} \\
+T_2 \\
+T_2 \\
-T_1 \\
-T_1
\end{array}
$$

Figure 13: The sequence of operations for a $-U_i$ is at left and that for a $+U_i$ is at right, for any $1 \leq i \leq n + 2$.

the other direction. We do not emphasize it, however, because it is not that relevant to our proof.) Moreover after executing all the operations of U_{ii}, the values of T_1 and T_2 go back to -2.

We claim that G_1 has a valid schedule if and only if G_2 has a valid schedule. The only-if part is straightforward. Suppose G_1 has a valid schedule. By Lemma 14 we know G_1 has a valid schedule executable by PAIRWISE. Note that we can execute the four operations of C_0 first, which decrease the value of both semaphores down to -2. Clearly the remaining $2n + 2$ chains of units can be completely pairwisely executed by following the sequence of corresponding operations in G_1 executed by PAIRWISE. Therefore G_2 has a valid schedule.

It takes some added work to prove the other direction of the above claim. A unit is *active* if its third operation is executed. An unit is *finished* (and thus inactive) if its fifth-to-last operation is executed. Suppose G_2 has a valid schedule. Consider the sequence of the units of G_2 that become active in the valid schedule. It follows from the following lemma that the corresponding sequence of operations of G_1 is a valid schedule of G_1. In fact it is "pairwise", since in the schedule each $-S_i$ is immediately followed by a $+S_i$.

Lemma 15 Consider the execution of a valid subschedule. 1) When there is no active unit, the next unit that becomes active must be a $-U_i$ for some $1 \le i \le n+2$. 2) Before that active $-U_i$ is finished, a $+U_i$ must become active. 3) No unit will become active unless these two active units are finished.

Proof At the beginning of the valid schedule, no unit is active. We show the first statement of the lemma holds. At this moment there are two $-T_1$'s and two $-T_2$'s available (in C_0). They are our only hope for activating any unit, since each unit is guarded by two $+T_1$'s and two $+T_2$'s. Assume for a contradiction that the first unit becoming active is a $+U_i$ for some $1 \le i \le n+2$. Note that as soon as the first $+U_i$ becomes active, at least two $+T_1$'s are already executed. Since at most two $-T_1$'s are executed so far, there is no way to activate any other unit. The execution thus cannot proceed.

When the first unit $-U_i$ becomes active, one can see that the second statement of the lemma holds by verifying the following.

- The active $-U_i$ will not be finished unless another unit becomes active, since otherwise the execution will be blocked by some $+T_2$'s.

- The next active unit must be a $+U_j$ for some $1 \le j \le n+2$, since otherwise the execution will be blocked by some $+T_2$'s.

- if $i < j$, the execution will be blocked by some $+T_1$'s. If $i > j$, the execution will be blocked by some $+T_2$'s. Therefore the next active unit must be a $+U_i$.

When those two units are active, in order to activate other units, we can only hope for the $-T_1$'s at the end of the active $+U_i$. In order to reach those $-T_1$'s, the preceding consecutive $+T_2$'s must be penetrated. Hence at least two $-T_2$'s at the end of the active $-U_i$ must be executed first. Therefore those two active units $-U_i$ and $+U_i$ must be finished before any other unit becomes active. This confirms the third statement of the lemma.

Note that as soon as the active $+U_i$ is finished (and so must be the active $-U_i$), the situation is exactly the same as the situation at the very beginning of the execution. Namely we have two $-T_1$'s and two $-T_2$'s available, which are again our only hope for activating any other units. Therefore all the above argument follows inductively. The lemma is proved. $\square$

5.4 Third Step

Let v be the first operation of the C_0 in G_2. Let w be the last operation of the C_{n+1} in G_2. We claim that $v \to w$ if and only if G_2 has a valid schedule. Note that if $\to$ is considered for valid schedules, as in [8], then the claim holds trivially, since v is always the first node in any valid (sub)schedule of G_2. As for the general $\to$, as defined in this paper, the if part is still straightforward for the same reason. It remains to prove the other direction of the claim.

Let X be a valid subschedule of G_2 in which v precedes w. Consider the sequence of the units of G_2 that become active while executing X. It follows from Lemma 15 that the corresponding sequence of operations of G_1 is a valid subschedule of G_1, which definitely contains the last $+S_\alpha$ of the C_{n+1} in G_1. Therefore G_1 has a valid schedule by Lemma 11 and Lemma 12. Finally it follows from the claim in Subsection 5.3 that G_2 has a valid schedule.

References

[1] Abdel-Wahab, H. M., "Scheduling with Application to Register Allocation and Deadlock Problems," University of Waterloo, PhD Thesis, 1976.

[2] Abdel-Wahab, H. M. & Kameda, T., "Scheduling to Minimize Maximum Cumulative Cost Subject to Series-parallel Precedence Constraints," *Operations Research* 26 (1978), 141–158.

[3] Abdel-Wahab, H. M. & Kameda, T., "On Strictly Optimal Schedules for the Cumulative Cost-Optimal Scheduling Problem," *Computing* 24 (1980), 61–86.

[4] Emrath, P. A., Ghosh, S. & Padua, D. A., "Event Synchronization Analysis for Debugging Parallel Programs," *Supercomputing '89* (November 1989), 580–588.

[5] Garey, M. R. & Johnson, D. S., *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[6] Helmbold, D. P. & McDowell, C. E., "A Class of Synchronization Operations that Permit Efficient Race Detection," *University of California at Santa Cruz Technical Report* (January 1993).

[7] Helmbold, D. P., McDowell, C. E. & Wang, J-Z., "Analyzing Traces with Anonymous Synchronization," *International Conference on Parallel Processing* (August 1990), II70–II77.

[8] Lu, H-I., Klein, P. N. & Netzer, R. H. B., "Detecting Race Conditions in Parallel Programs that Use One Semaphore," *Workshop on Algorithms and Data Structures* 3 (1993), 471–482.

[9] Netzer, R. H. B. & Ghosh, S., "Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization," *International Conference on Parallel Processing* (August 1992), II242–II246.

[10] Netzer, R. H. B. & Miller, B. P., "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," *International Conference on Parallel Processing* (August 1990), II93–II97.