Ordering Problems Approximated: Register Sufficiency, Single-Processor Scheduling and Interval Graph Completion

Ajit Agrawal, Philip Klein and R. Ravi

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-18
March 1991

Ordering problems approximated: register sufficiency, single-processor scheduling and interval graph completion

Ajit Agrawal * Philip Klein* † R. Ravi*

Abstract

In this paper, we give the first poly-time approximation algorithms for three problems in combinatorial optimization. The first problem is register sufficiency where, given a computation dag, we are required to find the minimum number of registers needed to compute it. This problem arises in compiler optimization, in the task of register allocation. The second problem is single-processor scheduling to minimize weighted sum of completion times, subject to precedence constraints. The third problem, interval graph completion, is finding a minimum-size interval graph containing the input graph as a subgraph. All of the above problems are NP-complete; our algorithms output solutions that are within a polylogarithmic factor of optimal. To achieve these bounds, we make use of a technique developed and first applied by Leighton and Rao [13], together with a technique of Hansen [6].

*Brown University

†Research supported by NSF grant CCR-9012357, NSF grant CDA 8722809, ONR, and DARPA, contract N00014-83-K-0146, and ARPA Order No. 6320, Amendment 1.

1 Introduction

1.1 Graph ordering problems

The first problem we consider is register sufficiency: Given a computation dag, how many registers are needed to compute it? This NP-complete problem arises in compiler optimization, in the task of register allocation. In this problem, an ordering is sought in the flow graph that uses the minimum number of temporary locations to store intermediate values. This problem is shown to be NP-complete in [16]. We present here a polynomial time algorithm that finds an order for computing an n-node dag such that the number of registers required is within $O(\log^2 n)$ factor of the optimal.

The single-processor scheduling problem, scheduling to minimize weighted sum of completion times, arises in a manufacturing process or computational process when we want to get everything done in a hurry, and it is more important that some tasks get done quickly than others (thus the weights). Numerous papers have been written on the solution of special cases and approaches to this problem [1, 2, 3, 7, 17, 18] and Lawler [10] showed that it is strongly NP-complete. We in fact give an approximation algorithm for a more general problem, in which the goal is to minimize the storage-time product for the process. This problem is intended to model a situation arising in manufacturing or computing in which storage is an expensive resource, and its use must be minimized.

The other problem we consider is that of finding a smallest interval graph that includes the input graph. This problem arises, e.g. in archeology [8] when one is trying to find a chronological model for data. Edges in the original graph correspond to, say, pairs of tools whose use overlapped in time; adding edges to the graph corresponds to assuming chronological overlap between tools without direct evidence. Finding a minimum interval graph completion, then, corresponds to finding a chronological model for tool use while making as few assumptions as possible. Unfortunately, the interval graph completion problem is NP-complete [4]. Moreover, no algorithm is known for approximately minimizing the number of edges whose addition yields an interval graph. As a first step towards finding such an algorithm, we have developed an algorithm that approximately minimizes (to within a log-squared factor) the number of edges in the completed graph. In [9], we gave and analyzed an approximation algorithm for minimum chordal completion, also based in part on [13], but the analysis was quite different.

2 Separators in Graphs

In this section, we define the notion of a separator which we use in this paper. The basic idea is to remove nodes or edges in order to split a graph into pieces, each of which is "small" with respect to the original graph in that its *node weight*—the sum of the weights of its nodes—is at most a fraction of the node weight of the original graph.

Defn: Suppose G is a graph with node weights. If G is undirected, we define a *b-bounded node separator* of G to be a set of nodes of G whose removal separates G into pieces, where the node weight of each piece is at most a fraction b of the total node weight of G. The cost of the separator is the total weight of the removed nodes. The *balance* of an edge/node separator that splits the graph into two pieces A and B as defined in [13], is the ratio of the smaller of the two pieces to the size of the whole graph. A *b-balanced edge (node) separator* of G is just a edge (node)

separator achieving a balance of b.

As a consequence of the approximate max-flow min-cut theorem of Leighton-Rao, one can find weighted edge- and node-separators in a graph. We list their results below.

Lemma 2.1 **([13, 12])** *For a graph (directed or undirected), there exists a polynomial algorithm to find a $\frac{1}{3}$-balanced edge-separator (node-separator) with cost within $O(\log n)$ factor of the optimal $\frac{1}{2}$-balanced edge-separator (node-separator).*

We use the above lemma in finding special types of separators in directed acyclic graphs. We define them in the corresponding sections where they are used.

3 Approximating Register Sufficiency

In this section, we present a polylogarithmic approximation to the *register sufficiency* problem.

3.1 Problem definition

Defn: Given a DAG G (where $|G| = n$) with its vertices numbered by a topological ordering τ, the *register cost at step* i is defined as the number of nodes in the set $\{1, \ldots, i\}$ that are tails of edges that go from the set $\{1, \ldots, i-1\}$ to the set $\{i, \ldots, n\}$. The *maximum register cost* of this ordering, denoted by MRC_τ, is the maximum of the register costs over all steps i. We shall refer to the minimum value of the maximum register cost achievable by any topological ordering for G as the *optimum register cost* M for G. The register sufficiency problem is to find an ordering τ such that $MRC_\tau = M$. This problem is shown to be NP-complete in [16]. We give a polynomial time algorithm that finds a topological ordering of G with its maximum register cost within a polylogarithmic factor of M.

3.2 Directed node separators in a DAG

We observe that one can find directed node separators in a DAG. That is, we find a partition, (L, X, R), of the nodes of G such that there are no edges of G between L and R, and all edges between X and R are directed from X to R.

Defn: The *node cut ratio* of a directed node separator, (L, X, R), is defined to be the ratio $|X| / \min(|L \cup X|, |R \cup X|)$. The *sparsest node cut ratio* of a graph is the smallest node cut ratio of any directed node separator in the graph.

Lemma 3.1 *Given a DAG G, we can find a directed node separator of G whose node cut ratio is within a factor of $O(\log n)$ of the sparsest node cut ratio.*

Proof: We construct an auxiliary directed graph G' from G with the property that we can recover a node-separator of G from an edge-separator of G'. We augment the DAG G to G' as follows. For each node v in G, we add two nodes v_i and v_o in G' with a directed edge from v_i to v_o of infinite cost and a reverse edge from v_o to v_i of unit cost. We call such reverse edges the *pseudo-edges.* For each original edge (u, v) in G, directed edges (u_o, v_i) and (v_i, u_o) of infinite cost are added (See Figure 1).

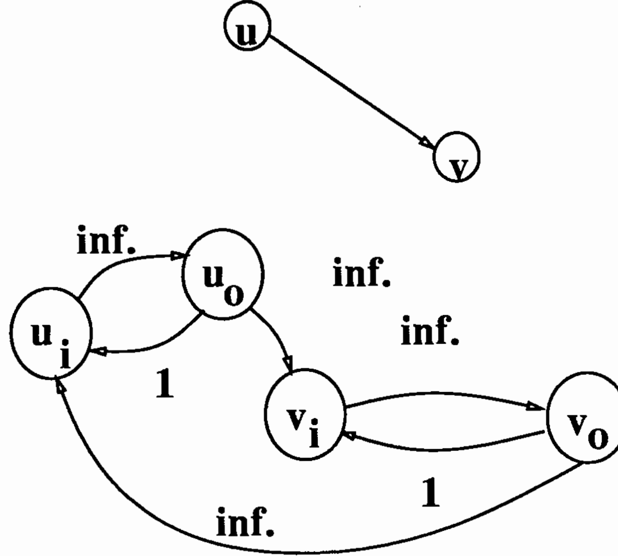

Figure 1: Augmentation of the DAG G to a strongly connected graph G'. Each original node v in G is split into two nodes v_i with all the incoming edges of v and v_o with the outgoing edges from v. We add a forward edge from v_i to v_o of infinite cost and a reverse edge of unit cost. Also, for each original edge (u, v) in G, we add a forward edge (u_o, v_i) and a reverse edge (v_i, u_o), both of infinite cost. Informally, this way, we can recover a node separator in G from a directed edge separator in the strongly connected graph G'.

The algorithm in [13] finds a directed edge-separator in G' where the ratio of the cost of the separator edges divided by the number of nodes on the smaller side is within a factor of $O(\log n)$ of the minimum value. We call this minimum value the *sparsest edge cut ratio.*

Any optimal or near-optimal edge separator for G' will only contain pseudo-edges, hence we can recover a node separator in G from these pseudo-edges. Since any node separator with a node cut ratio of n_r in G can be mapped to an edge separator in G' with an edge cut ratio of at most n_r, the sparsest edge cut ratio in G' is no more than the sparsest node cut ratio in G. Moreover, for any edge separator in G' with an edge cut ratio of e_r, the node separator recovered from it has a node cut ratio of no more than $2e_r$ in G. Hence from an approximate edge separator algorithm for G' with a performance guarantee of $c \log n$, one can find an approximate directed node separator in G with a performance guarantee of $2c \log n$. $\square$

3.3 The algorithm

We now describe the algorithm for the register sufficiency problem.

We find an approximate sparsest directed node separator (L, X, R) in G as outlined in Lemma 3.1. If $|X| / \min(|L|, |R|)$ is greater than $1/\log n$ then we output an arbitrary topological ordering for G. Otherwise, we partition G into $L \cup X$ and R. We recursively order each of these subgraphs, and output the topological ordering of G consisting of the recursively produced order of $L \cup X$ followed by the recursively produced order of R.

3.4 Performance guarantee

We now argue that this algorithm produces the desired result. First, it is easy to see that the following fact holds.

Fact 3.1 *The ordering τ defined by the above algorithm is a topological ordering.*

Now we proceed by showing that τ is a *good* topological ordering.

Theorem 3.1 *Given an n-node DAG G, one can in polynomial time find a topological ordering τ of G such that MRC_τ is within a factor of $O(\log^2 n)$ of the optimum register cost M for G.*

Proof: Consider the first cut, (L, X, R), that is used in the algorithm above to produce τ. Notice that if the algorithm recurses the schedule τ needs to use only as many registers as the size of X plus the maximum of the number of registers that τ needs for evaluating $L \cup X$ or to evaluate R.

Thus we can use the following recurrence to estimate the performance bound when the algorithm recurses.

$$S(n) \leq |X| + \max(S(|L \cup X|), S(|R|)) \quad (1)$$

If the algorithm does not recurse, we use the trivial bound of $S(n)$ being at most n.

We proceed by bounding the size of $|X|$ in terms of M. To do this we consider an optimal ordering, τ_{opt} of G. We form a partition, (A, B), of the nodes of G where A consists of the first $n/2$ nodes in the ordering τ_{opt}. Now consider the set of nodes in A with a neighbor in B: these form a directed node separator of G. Clearly, a register must be used for each of these. Thus, there are no more than M such nodes. Thus, there exists a node separator of G with sparsest cut cost of at most $M/|A| = M/(n/2)$. Now recall that (L, X, R) has sparsest cut cost of $O(\log n)$ times optimal. Thus,

$$\frac{|X|}{\min(|L \cup X|, |R \cup X|)} \leq c \log n \frac{M}{n/2}. \quad (2)$$

When the algorithm does not recurse, $\frac{\min(|L \cup X|, |R \cup X|)}{|X|}$ is at most $\log n + 1$. Thus from (2), n, and hence $S(n)$, is at most $2cM \log n(\log n + 1)$, which is $O(M \log^2 n)$.

Now we consider the case where the algorithm does recurse. With the above bound on $|X|$, we can rewrite recurrence (1) as

$$\begin{aligned} S(n) \;\; &\leq \;\; 2cM \log n \frac{\min(|L \cup X|, |R \cup X|)}{n} \\ &\quad + S(\max(|L \cup X|, |R|). \end{aligned}$$

When the algorithm recurses, we have $|X| \leq \frac{\min(|R|, |L|)}{\log n}$, so we can rewrite the above inequality as

$$\begin{aligned} S(n) \;\; &\leq \;\; 2c' M \log n \frac{\min(|L \cup X|, |R|)}{n} \\ &\quad + S(\max(|L \cup X|, |R|). \end{aligned}$$

where c' is no more than $c(1 + \frac{1}{\log n})$.

We further simplify the above equation to be

$$S(n) \leq 2rc'M\log n + S((1-r)n), \tag{3}$$

where

$$r = \frac{\min(|L \cup X|, |R|)}{n} = (1 - \frac{\max(|L \cup X|, |R|)}{n}).$$

Finally, we note that at most M registers are needed to evaluate any subgraph of G. Inductively assuming that $S(n')$ is $c''M\log^2 n'$, we can infer from (3) that $S(n) \leq c''M\log^2 n$ for an appropriate constant c''. This is elaborated below.

$$\begin{aligned} S(n) &\leq 2Mrc'\log n + S((1-r)n) \\ &\leq 2Mrc'\log n + c''M\log^2 n + 2c''M\log(1-r)\log n \\ &\quad +2c''M\log^2(1-r) \\ &\leq c''M\log^2 n + M\log n(2rc' + c''\log(1-r)) \\ &\leq c''M\log^2 n + M\log n(2rc' - c''r) \end{aligned}$$

Thus, $S(n)$ in recurrence 1 is at most $c''M\log^2 n$ if $c'' \geq 2c'$. □

4 Approximating scheduling problems

In this section, we present guaranteed approximation algorithms for two NP-complete problems related to optimizing single processor schedules. In particular, we look at the following two problems.

1. Finding a single-processor schedule that minimizes weighted completion time.
2. Finding a single-processor schedule that minimizes the total storage-time units used in a given computation.

The performance bounds are poly-logarithmic in the total weight of the input graph. We elaborate on these below.

4.1 Problem definitions

We need to give more detailed definitions for the scheduling problems. An instance of single-processor scheduling to minimize weighted sum of completion times consists of a directed acyclic graph G, and an execution time $\ell(v)$ and a weight $w(v)$ for each node v. A node v represents a task that takes time $\ell(v)$ to perform, and an edge (u, v) represents the constraint that u must be performed before v. A schedule consists of an ordering of the nodes that is consistent with the precedence edges, i.e. a topological ordering. We assume that the execution of tasks begins at time 0, so the completion time for the ith task in the schedule is the sum of the execution times of the first i tasks. The goal is to minimize the weighted sum of completion times, i.e. the sum, over all tasks v, of $w(v)$ times v's completion time.

Theorem 4.1 *There is a polynomial-time algorithm to minimize the weighted sum of completion time to within a factor of $O(\log n \log L)$, where L is the sum of execution times, and n is the number of nodes.*

If the weights are polynomially bounded in n, then it follows using standard techniques that the $\log L$ in the performance guarantee can be replaced by $\log n$.

To prove Theorem 4.1, we show that this problem can be reduced to another scheduling problem, scheduling to minimize time-storage, subject to precedence constraints. The input consists of a dag with nodes and arcs, where nodes and arcs are labeled with nonnegative integers. A node represents a task that needs to be performed, and an arc (x, y) represents an output of task x that is an input to task y. As before, the number assigned to a node is the processing time required to perform the task, assuming all its inputs are available. The number assigned to an arc (x, y) is a measure of the amount of storage needed to hold the output of x that is used as an input to y. As before, a schedule is a topological ordering of the tasks, i.e. an ordering consistent with the arcs.

A schedule determines costs on the arcs; the cost of an arc (x, y) is the storage required by the arc (the arc's weight) times the amount of time that storage is needed (the time required for all tasks from x through y). The total cost determined by the schedule is the sum of all the arc-costs. The problem, then, is to choose a schedule that minimizes the total cost.

Theorem 4.2 *There is a polynomial-time algorithm to minimize the total storage-time cost to within a factor of $O(\log n \log L)$, where L is the sum of execution times, and n is the number of nodes.*

Again we can replace $\log L$ by $\log n$ if the amount of storage needed by each edge is polynomial in n.

4.2 Techniques

Both of the problems we consider are closely related to the problem of finding an optimal linear arrangement. This is the problem of ordering a graph's nodes from 1 to n so as to minimize the sum over all edges $\{v, w\}$ of the number of nodes between v and w. Optimal linear arrangement was shown to be NP-complete by Garey, Johnson, and Stockmeyer [5]. Hansen has shown [6] how to approximately solve a more general problem, embedding the node-set of a graph in a d-dimensional grid so as to approximately minimize the weighted sum of edge-distances. In obtaining this result, one of several in the paper, he uses a simple lemma (Claim 14) relating the cost of the embedding to the minimum size of a separator in the graph. He combines this lemma with a recently developed algorithm of Leighton and Rao [13] for finding an approximately minimum balanced separator.

This paper consists in further applying these two techniques. Hansen's Claim 14 finds its way into our paper in the form of Lemmas 4.3 and 5.1. In the application to scheduling, we define a kind of balanced separator especially suitable for separating directed acyclic graphs, and we show a reduction from the problem of finding an approximately minimum such separator to an algorithm of Leighton and Rao for separating a strongly connected graph. Furthermore, in this application the nodes are weighted, and our goal is to separate the graph into pieces of small weight. As observed in [11], it is an easy matter to generalize Leighton and Rao's proof to handle such separation by weight. We combine this observation with our reduction, and achieve an algorithm for exactly the kind of weight-balanced dag separators we need for the scheduling algorithm.

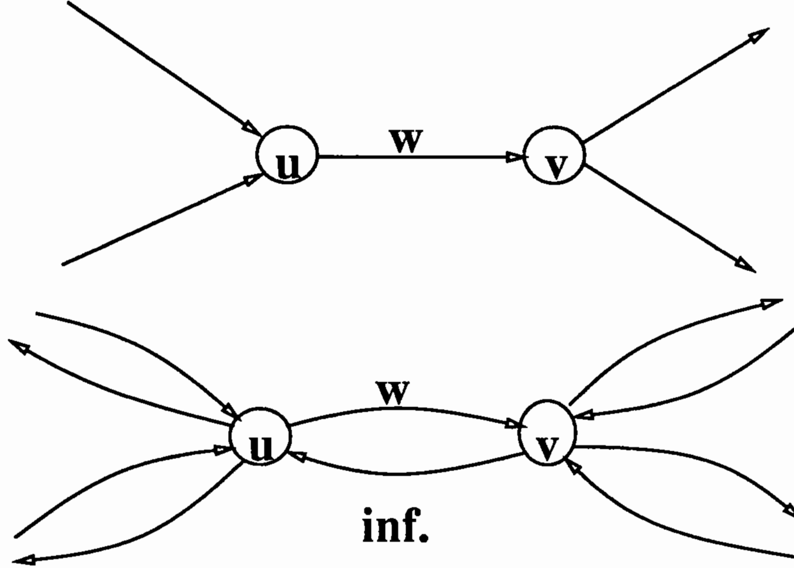

Figure 2: Augmentation of the DAG G to a strongly connected graph G' by adding a reverse edge of infinite cost for every original edge (u, v) of cost w in G.

4.3 DAG edge-separators

Defn: For a directed acyclic graph G, we define a *dag edge-separator* to be a partition of the nodes into two sets, A and B, such that all edges between A and B go from A to B. The cost of the separator is the total weight of edges from A to B. The separator is said to be b-balanced if A and B each have node weight at least a fraction b of G's node weight.

We show that a small separator of this form can be found using the techniques of [13]. In particular, we show a reduction from finding a small separator of this form to finding a small separator of another form, defined by Leighton and Rao. Our reduction is closely related to one we presented in [9].

Defn: Leighton and Rao define an edge separator for a strongly connected graph G' as a three-way node partition $(S, T, \overline{S \cup T})$, where the edges that are considered to comprise the separator are all edges that leave S or enter T. The cost of the separator is defined to be the sum of the weights of the edges comprising it. They define such a partition to be *b-balanced* if the node weight of $S \cup T$ and the node weight of $\overline{S \cup T}$ are each *at least* a fraction b of the total node weight of G'. They also prove the following lemma.

Lemma 4.1 *There is a polynomial-time algorithm to find a $\frac{1}{4}$-balanced edge-separator with cost within a $O(\log n)$ factor of cost of the minimum-cost $\frac{1}{3}$-balanced edge-separator.*

We now derive from Lemma 4.1 a result about finding edge-separators in dags.

Lemma 4.2 *Given a DAG G with weights on the nodes and edges, we can find a $\frac{1}{8}$-balanced dag edge-separator of G whose cost is within a factor of $O(\log n)$ of the minimum-cost $\frac{1}{3}$-balanced dag edge-separator of G where n is the total number of nodes in G.*

Proof:

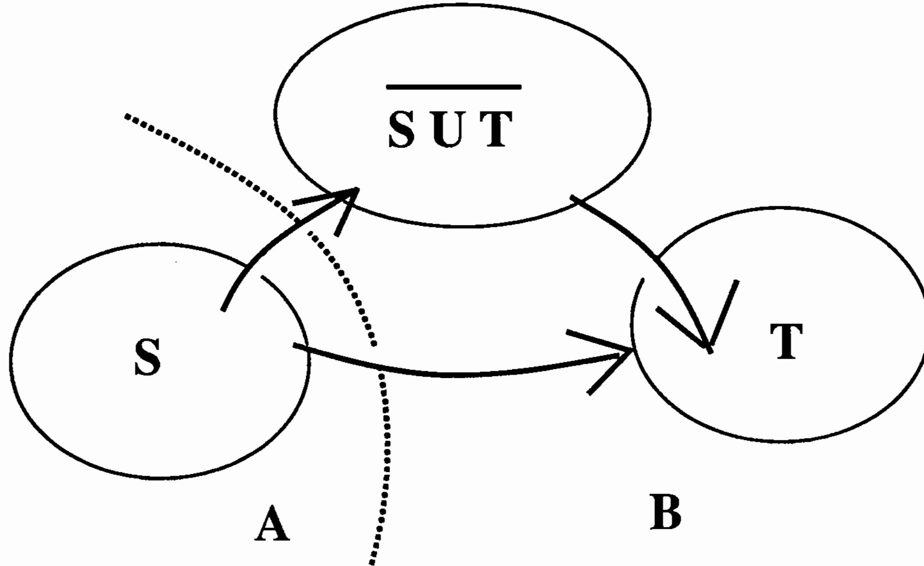

Figure 3: The augmented graph G' is separated by the algorithm of Leighton and Rao [13] into three pieces S, T and $\overline{S \cup T}$, where the edges in the separator in G' are all the edges that leave S or enter T as shown above Since both pieces $S \cup T$ and $\overline{S \cup T}$ are $\frac{1}{4}$-balanced, assuming that S has more weight than T, then the partition A and B is $\frac{1}{8}$-balanced and forms a dag separator of G.

We transform the DAG G to a strongly connected graph G' as follows. The nodes of G' are the same as the nodes of G. An edge (u, v) in G of weight w is replaced in G' by a pair of edges, an *original* edge (u, v) of weight w and a *reverse* edge (v, u) of infinite weight (See Figure 2).

We now find an approximate $\frac{1}{4}$-balanced edge-separator $(S, T, \overline{S \cup T})$ in G' using the algorithm of Lemma 4.1 (See Figure 3). Note that no reverse edges can be in this separator since they have infinite weight. Hence, the edges in the separator are all reverse edges. Without loss of generality, suppose S has at least as much node weight as T. Hence, its weight is at least a fraction 1/8 of the total node weight of G. In this case, we choose A to be S, and B to be $T \cup (\overline{S \cup T})$. Then the partition (A, B) in the original graph G is a $\frac{1}{8}$-balanced directed edge separator. The cost of the directed edge separator in G is the same as the cost of the $\frac{1}{4}$-balanced edge-separator in G', which in turn is at most $O(\log n)$ times the minimum cost of a $\frac{1}{3}$-balanced edge-separator in G.

It remains only to point out that for any $\frac{1}{3}$-balanced dag edge-separator (A, B) in G, there corresponds a $\frac{1}{3}$-balanced edge-separator in G, namely $(A, \emptyset, B)$. □

4.4 Minimizing storage-time units

Let $G = (V, E)$ be a DAG. The nodes represent tasks to be scheduled on a single processor. The time required to execute the task t is denoted by $\ell(t)$. Each edge $e = (u, v)$ has a weight $w(e)$ associated with it. This weight represents the number of units of storage required to save the intermediate results generated by task u until they are consumed at task v. Let $\tau = v_1, \ldots, v_n$ be a topological ordering of G. We denote the storage-time cost for this ordering by C_τ. We define

C_τ as

$$C_\tau = \sum_{\text{all edges } e} s(e)w(e)$$

where $s(e)$ for an edge $e = (v_i, v_j)$ is exactly the sum of the execution times of all tasks ordered between tasks v_i and v_j, inclusive. In other words,

$$s(e) = \sum_{k=i}^{j} \ell(v_k)$$

The problem is to find the topological ordering τ_{opt} that minimizes the storage-time cost over all orderings. We call $s(e)$ and $s(e)w(e)$ respectively the *stretch* and the *weighted stretch* of the edge e.

The main result of this section is as follows.

Theorem 4.3 *Given a DAG G whose nodes are assigned task execution times, and whose edges are assigned storage costs, there is a polynomial-time algorithm to find an ordering of the tasks whose storage time cost is within a factor of $O(\log n \log L)$ of the optimal where L is the sum of the execution times of all the tasks in G, and n is the total number of nodes in G.*

4.4.1 A lower bound

We first derive a lower bound for the cost of a topological ordering. Consider a topological ordering $\tau = v_1, \ldots, v_n$.

Defn: We define the cut S_i to be the set of edges going from nodes v_1 through v_i to the rest of the nodes. We define the cost of this cut, C^i to be the sum of the edge weights of all the edges in the cut, multiplied by the weight of the vertex v_i.

Observation 4.1 *For any topological ordering τ the sum of the weighted stretches of all the edges, and hence the storage-time cost of τ, is at least $\sum_{i=1}^{n} C^i$.*

We use this observation to obtain a useful lower bound on the cost of the minimum storage-time schedule. This lemma is analogous to Claim 14 of [6].

Lemma 4.3 *For a computation DAG G with total completion time L such that the length of each task is at most $L/6$, the optimal storage-time cost is at least $\Omega(LB)$ where B is the weight of the minimum $\frac{1}{3}$-balanced dag edge-separator of G.*

Proof: Consider the topological ordering τ_{opt} that minimizes the storage-time product in G. Let i be the minimum index such that $\sum_{m=1}^{i} \ell(v_m)$ is at least $\frac{1}{3}$ of L, and let j be the maximum index such that $\sum_{m=j}^{n} \ell(v_m)$ is at least $\frac{1}{3}$ of L. For any k between i and $(j-1)$ inclusive, the cut S_k is a $\frac{1}{3}$-balanced dag edge-separator, and hence has weight at least B, where B is the cost of the minimum $\frac{1}{3}$-balanced dag edge-separator. Since we assumed that each node has weight at most $L/6$, it follows that $\sum_{k=j}^{n} \ell(v_k) \leq L/3 + L/6$ and $\sum_{k=i}^{j-1} \ell(v_k) \geq L/6$ Thus, we have

$$C_{opt} \geq \sum_{k=1}^{n} C^k \ell(v_k) \geq B \sum_{k=i}^{j-1} \ell(v_k) \geq BL/6$$

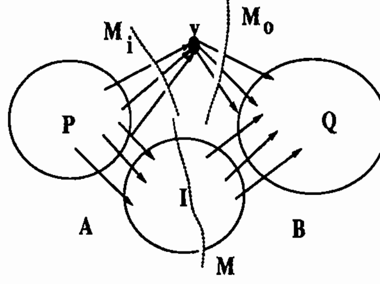

Figure 4: If there is a node of weight more than a sixth of the total node weight in the graph G, then we partition the nodes of $G - \{v\}$ into pieces P, Q and I that are respectively before, after, and incomparable to v in the partial order represented by the DAG G. We then contract P and Q to single nodes p and q respectively and find a minimum weighted DAG edge-separator $(\hat{A}, \hat{B})$ of weight M in $\hat{G}_v$. We use M_i and M_o to denote the weights of the incoming and outgoing edges of v respectively. We can then derive a minimum v-DAG edge-separator (A, B). Namely, we let A equal $\hat{A} \cup P - \{p\}$ and B equal $\hat{B} \cup Q - \{q\}$.

□

Now, let v be any node in G, let P be the set of ancestors of v (nodes that can reach v), and let Q be the set of descendents of v. Let $\hat{G}_v$ be the graph obtained from G by deleting v, contracting P to a single node p, and contracting Q to a single node q. We find a minimum DAG edge-separator $(\hat{A}, \hat{B})$ where $p \in \hat{A}$ and $q \in \hat{B}$. Such a separator can be found using a min-cut computation in $\hat{G}_v$. Let A be the set of nodes of G in $\hat{A}$, together with P, and let B be the set of nodes of G in $\hat{B}$, together with Q. We call (A, B) the *minimum v-DAG edge-separator.* Intuitively, (A, B) is a *DAG* edge-separator in $G - \{v\}$. In the next lemma, we give a simple lower bound on the storage-time cost of G. This lower bound is particularly useful in the case where there is a node of large weight. The lemma is illustrated in Fig. 3.

Lemma 4.4 *Let M be the minimum v-DAG edge-separator in $\hat{G}_v$. Let M_i and M_o be the total weight of the incoming edges and outgoing edges of v, respectively. Then $(M + M_i + M_o)$ times the weight of v is a lower bound on the storage-time cost of G.*

4.4.2 The algorithm

We now give the algorithm for finding a topological ordering of a DAG G that approximately minimizes the storage-time cost. The algorithm is as follows. If G consists of a single node, the algorithm is trivial. Otherwise, we proceed recursively as follows.

If every node is of weight at most a factor $\frac{1}{6}$ of the total node weight of the graph L, then we find an approximately minimum $\frac{1}{8}$-balanced dag edge-separator (A, B) in G as outlined in Lemma 4.2. The topological ordering of G consists of the nodes of A, recursively ordered, followed by those in B, also recursively ordered.

If there is a vertex v in G of weight greater than $L/6$, then we make use of Lemma 4.4 and find a minimum v-DAG edge-separator (A, B). The topological ordering of G consists of the recursive topological ordering of A, the node v, and the recursive topological ordering of B.

4.4.3 Performance Guarantee

Let us introduce some notation for convenience. We define a stage of the algorithm as follows. Let $G_1, \ldots, G_k$ be the subgraphs of G at the beginning of stage i. At the first stage there is only one such graph and that is G itself. The subgraphs at the beginning of the stage $i+1$ are the subgraphs $A_1, B_1, A_2, B_2, \ldots, A_k, B_k$, where A_j and B_j are the graphs found by decomposing G_j for recursively ordering its nodes. For each such graph G_j, some of its edges are removed during the decomposition. These edges go between the node sets A_j and B_j if G_j has no large weight node. If G_j has a large weight node v, then the edges either go between the node sets A_j and B_j, between the nodes in A_j and the node v, or between the node v and the nodes in B_j.

Observation 4.2 *Each edge in G is removed once and exactly once by the algorithm.*

Lemma 4.5 *The number of stages in the above algorithm executed on G is $O(\log L)$, where L is the sum of the execution times of the tasks in G.*

Proof: At each stage, for a graph G_j with node weight L_j, the decomposed node sets A_j and B_j found by the algorithm have weight at most $\frac{7}{8}L_j$. Thus the node weights of each of the subgraphs at stage i is at most $(\frac{7}{8})^i L$, where L is the node weight of G. After at most $\log_{\frac{8}{7}} L$ stages, each subgraph consists of at most one node and the algorithm terminates. □

Lemma 4.6 *The sum of the weighted stretches of the edges removed at a stage is at most $O(C_{\tau_{opt}} \log n)$, where $C_{\tau_{opt}}$ is the storage time product for the optimal topological ordering τ_{opt}, and n is the number of nodes in the graph.*

Proof: Let $G_1, \ldots, G_k$ be the subgraphs at stage i. Consider a subgraph G_j with node weight L_j.

If G_j does not have a high-weight node, then by lemma 4.3 the optimal storage time product C^j_{opt} for ordering its nodes is at least $B_j L_j/6$, where B_j is the weight of the minimum $\frac{1}{2}$-balanced edge separator of G_j. By lemma 4.2, the weight of the edges removed from G_j is at most $cB_j \log n_j$, where n_j is the number of nodes in G_j, and c is the constant for the separator algorithm. Hence the sum of their stretches in the ordering given by the algorithm can be at most $cL_jB_j \log n_j$, and hence at most $6cC^j_{opt} \log n$.

Now suppose G_j has a node of weight at least $\frac{L_j}{6}$. In this case, we use Lemma 4.4, which is illustrated in Fig. 3. By Lemma 4.4, C^j_{opt} is at least $(M_i + M_o + M)\frac{L_j}{6}$. The sum of the weighted stretches of the edges removed from G_j is at most $(M_i + M_o + M)L_j$ and hence is at most $6C^j_{opt}$.

Since $\sum_{j=1}^{k} C^j_{opt}$ is at most $C_{\tau_{opt}}$, it follows that the sum of stretches of the edges removed at a stage is at most $\mathrm{O}\left(C_{\tau_{opt}} \log n\right)$. □

Theorem 4.3 follows from Lemmas 4.5 and 4.6.

4.5 Minimizing weighted completion time

We describe the problem of minimizing the weighted completion time of a DAG G. The nodes of G represent tasks. The execution time for the task t is denoted by $l(t)$, and the weight of the task is denoted by $w(t)$. For an ordering $\tau = \{v_1, \ldots, v_n\}$, the weighted completion time is given by $\sum_{i=1}^{n} w(v_i)\sigma(v_i)$, where $\sigma(v_k)$ denotes the time at which the task for the node v_k is completed in the given ordering. We assume that the execution of the first task begins at time 0, so $\sigma(v_k)$

is given by $\sum_{i=1}^{k} l(v_i)$. The objective is to find a topological ordering of G such that the weighted completion time is minimum over all such orderings. The problem is known to be NP-complete [4].

We reduce the above problem to an instance of minimizing the storage-time product. We then use the results of the previous sections to give a polynomial time algorithm to find a schedule whose cost is within a multiplicative factor of $O(\log n \log L)$ of the optimal, where L is the sum of the lengths of all the tasks in G.

4.5.1 Reduction to Minimizing Storage-Time product

From the given DAG G representing the computation, we build an augmented DAG G' as follows. We add a new node s to G and add directed edges from s to each of the vertices of G. We then assign weights to the nodes and edges of G' as follows. Each original node v, is allotted a weight equal to its execution time $l(v)$. The added node s is assigned weight 0. All the original edges of G are weighted 0. For every node t, the added edge (s, t) is assigned weight $w(t)$.

It can easily be checked that the value of the storage-time product for any ordering of G' is exactly the value of the weighted completion time of G for the same ordering of the nodes as in $G' - s$. This implies that approximating the minimum storage-time product for G' yields an approximation for the weighted completion time for G with the same performance guarantee. Thus we have the following theorem.

Theorem 4.4 *There is a polynomial time algorithm such that given a task DAG G, it produces a topological ordering of the nodes of G such that the weighted completion time for the ordering is within a multiplicative factor of $O(\log n \log L)$ of the optimal over all orderings, where n is the total number of nodes in G, and L is the sum of the execution times of the tasks in G.*

5 Interval Graph Completion

An interval graph is a graph whose vertices can be mapped to distinct intervals in the real line such that two vertices in the graph have an edge between them iff their corresponding intervals overlap. The interval graph completion problem consists in finding a smallest interval graph that contains the input graph as a subgraph, and has the same nodes as the input graph. This problem was proved to be NP-Complete [4].

Fact 5.1 *[14] There exists an ordering of the nodes in any interval graph such that if a node u with number i has an edge to a node v with number j, for i less than j, then every node with number between i and j has an edge to v.*

We shall refer to such an ordering as the "interval graph ordering." An ordering of the nodes of the input graph induces an interval graph containing the input graph as a subgraph, namely the graph obtained by adding edges as needed until the condition of 5.1 holds. This augmentation can be done in time proportional to the number of edges in the augmented graph. Our algorithm for interval graph completion will output an ordering such that the total number of edges in the augmented graph is small with respect to the optimum.

Let G be the input graph, and let G_{opt} be a smallest interval graph containing G. We shall start by giving a lower bound for the number of edges in G_{opt}. We then present an ordering of

the nodes of G such that the number of edges in its augmented interval graph is no more than a polylog factor of the number of edges in G_{opt}.

5.1 Lower Bound

Lemma 5.1 *If B is the size of the optimal $\frac{2}{3}$-bounded node-separator for G, then G_{opt} has $\Omega(Bn)$ edges.*

Proof: Let interval graph ordering of G_{opt} be $\tau = v_1, \ldots, v_n$. For each i, consider the set of nodes $V_i = \{v_1, \ldots, v_i\}$. Let the set of neighbors of V_i not in V_i be N_i, and suppose it has size C_i. For i between $n/3$ and $2n/3$, each of the node sets N_i is a $\frac{2}{3}$-bounded node-separator and hence C_i must be at least B for each such cut. Moreover, every node in N_i is adjacent to some node in V_i. Hence by fact 5.1 it follows that the node v_i must have edges in G_{opt} to each of the nodes in N_i. Thus the total number of edges in G_{opt} must be at least $\sum_{i=1}^{n} C_i$ which is at least $\sum_{i=\frac{n}{3}}^{i=\frac{2n}{3}} C_i$ and hence at least $Bn/3$. □

5.2 Algorithm

We now give the algorithm for ordering the nodes of G. If G consists of at most two nodes, we use any ordering. Otherwise, we find a $\frac{3}{4}$-bounded node separator of G using the algorithm of Lemma 2.1, order each of the pieces recursively, and pick any order between the pieces. The nodes in the separator are then ordered arbitrarily after all of the pieces.

Let G^* be the graph obtained from G by adding edges to make the resulting ordering an interval graph ordering. We show in the next subsection that the number of edges in G^* is small.

5.3 Performance Guarantee

Lemma 5.2 *The number of edges in G^* is no more than a $O\left(\log^2 n\right)$ factor of C_{opt}, the number of edges in G_{opt}.*

Proof:

Let us define the concept of stages as before. Let the ith stage consist of subgraphs $G_1, \ldots, G_k$. The first stage consists of the graph G. Let S_j be the the node separator for G_j found by the algorithm. Let $G_{j1}, \ldots, G_{jm_j}$ be the subgraphs of G_j after removing S_j. Then the stage $i+1$ consists of the subgraphs $G_{11}, \ldots, G_{1m_1}, \ldots, G_{k1}, \ldots, G_{km_k}$.

Let the optimally augmented interval graph for G_j have C_j edges. By lemma 5.1 C_j is at least $B_j n_j/3$, where B_j is the optimal $\frac{1}{3}$ balanced node separator of G_j, and n_j is the total number of nodes in G_j. All the edges in the interval graph produced by the algorithm must go between the pieces of G_j and the node separator of G_j found by the algorithm. The total number of such edges is at most $x_j n_j$, where x_j is the size of the node separator of G_j found by the algorithm. By Lemma 2.1, x_j is at most $cB_j \log n_j$, where c is the constant in the guarantee of the algorithm. By Lemma 5.1, therefore, the total number of edges added between the pieces of G_j is at most $3c \log n$ times C_j. Since the value of $\sum_{j=1}^{k} C_j$ is at most the number of edges in G_{opt}, it follows that the total number of edges added to the interval graph at any stage is at most $3c \log n C_{opt}$.

Moreover, since the sizes of the subgraphs go down by a factor of $3/4$ at each stage, it follows that the total number of stages is $O(\log n)$. This implies the lemma. □

6 Acknowledgements

We thank Satish Rao for his observations that led to the current form of the result claimed for the Register sufficiency approximation. He is one of the co-authors of our paper [9] which contains this result. The other two results are to appear in the International Colloquium on Automata, Languages and Processing '91 [15].

References

[1] D. Adolphson and T. C. Hu, "Optimal linear ordering", *SIAM J. Appl. Math. 25* (1973), pp. 403-423.

[2] K. R. Baker, "Single machine sequencing with weighting factors and precedence constraints," unpublished paper (1971).

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling* (1967), Addison-Wesley, Reading, Massachusetts.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco (1979).

[5] M. R. Garey., D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theor. Comput. Sci. 1* (1976), pp. 237-267.

[6] Mark D. Hansen, "Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems," *Proceedings, 30th Symposium on Foundations of Computer Science* (1989), pp. 604-609.

[7] W. A. Horn, "Single machine job sequencing with treelike precedence ordering and linear delay penalties," *SIAM J. Appl. Math.23* (1972), pp. 189-202.

[8] D. G. Kendall, "Incidence matrices, interval graphs, and seriation in archaeology," *Pacific J. Math. 28* (1969), pp. 565-570.

[9] P. Klein, A. Agrawal, R. Ravi, and S. Rao, "Approximation through multicommodity flow", *31st Annual Symp. on Foundations of Comp. Sci.,* (1990), pp. 726-737.

[10] E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints," *Annals of Discrete Math. 2* (1978), pp. 75-90.

[11] F. T. Leighton, Fillia Makedon, Serge Plotkin, Clifford Stein, Eva Tardos, Spyros Tragoudas, "Fast approximation algorithms for multicommodity flow problems," unpublished manuscript (1990).

[12] F. T. Leighton, F. Makedon, and S. Tragoudas, personal communication, 1990

[13] F. T. Leighton and S. Rao, "An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms", *29th Symposium on Foundations of Computer Science* (1988), pp. 422-431.

[14] G. Ramalingam, and C. Pandu Rangan, "A unified approach to domination problems in interval graphs", *Information Processing Letters, vol. 27* (1988), pp. 271-274.

[15] R. Ravi, A. Agrawal, P. Klein, "Ordering problems approximated: single-processor scheduling and Interval graph completion," To appear in the *International Colloquium on Automata, Languages and Processing '91*, (July 1991, Spain).

[16] R. Sethi, "Complete register allocation problems", *SIAM J. Comp. 4* (1975), pp. 226-248.

[17] J. B. Sidney, "Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs," *Operations Res. 22* (1975), pp. 283-298.

[18] W. E. Smith, "Various optimizers for single-stage production," *Naval Res. Logist. Quart. 3* (1956), pp. 59-66.