ON KNAPSACKS, PARTITIONS, AND A NEW DYNAMIC PROGRAMMING TECHNIQUE FOR TREES

by

D. S. Johnson and K. A. Niemi

Computer Sciences Technical Report #399

October 1980

ON KNAPSACKS, PARTITIONS, AND A NEW DYNAMIC PROGRAMMING TECHNIQUE FOR TREES

by

D. S. Johnson*
Bell Laboratories
Murray Hill, New Jersey 07974

K. A. Niemi
Bell Laboratories
Piscataway, New Jersey 08854

Abstract

Let G be an acyclic directed graph with weights and values assigned to its vertices. In the Partially Ordered Knapsack problem we wish to find a maximum-valued subset of vertices whose total weight does not exceed a given knapsack capacity, and which contains every predecessor of a vertex if it contains the vertex itself. We consider the special case where G is an out-tree. Even though this special case is still NP-complete, we observe how dynamic programming techniques can be used to construct pseudo-polynomial time optimization algorithms and fully polynomial time approximation schemes for it. In particular, we show that a non-standard approach we call "left-right" dynamic programming is better suited for this problem than the standard "bottom-up" approach, and we show how this "left-right" approach can also be adapted to the case of in-trees and to a related tree partitioning problem arising in integrated circuit design. We conclude by presenting complexity results which indicate that similar success cannot be expected with either problem when the restriction to trees is lifted.

---

*Work of this author partially supported by the Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

1. Introduction

Suppose we are given an acyclic directed graph $G = (V,A)$, a function $w: V \to Z_0^+$ assigning a non-negative weight to each vertex,* a function $p: V \to Z_0^+$ assigning a value to each vertex, and an integer knapsack capacity $B \geq \max\{w(v): v \in V\}$. Let us say that a subset $V' \subseteq V$ is closed under predecessor if $v \in V'$ and $(u,v) \in A$ imply that $u \in V'$. In the Partially Ordered Knapsack problem we wish to find a subset $V' \subseteq V$ which is closed under predecessor, such that $w(V') \equiv \sum_{v \in V'} w(v) \leq B$ and $p(V') \equiv \sum_{v \in V'} p(v)$ is maximized.

In the Graph Partitioning problem we are given an undirected graph $G = (V,E)$ with weights $w: V \to Z_0^+$ on its vertices and values $p: E \to Z_0^+$ on its edges, and a knapsack capacity $B \geq \max\{w(v): v \in V\}$. We wish to find a partition $V = V_1 \cup V_2 \cup \ldots \cup V_m$ of V such that $w(V_i) \leq B$, $1 \leq i \leq m$, and such that the total value of edges connecting vertices in different sets, $p(\{\{u,v\} \in E: \{u,v\} \not\subseteq V_i, 1 \leq i \leq m\})$, is minimized. See Figure 1 for a sample instance of both these problems.

The Partially Ordered Knapsack problem can be viewed as modelling an investment situation in which the vertices represent potential investments, their weights are costs and their values are expected profits, and in which certain investments can be made only if other ones have been made previously [4]. For example, in strip mining, the miner must decide where to do his digging, knowing that upper strata must be excavated before the ones beneath them can be reached. The Graph

---

*We denote by Z^+ the set of positive integers and by Z_0^+ the set of non-negative integers.

Partitioning problem has application to the partitioning of networks or electronic circuits to minimize delay or interconnections [7].

This paper follows [4,7] in concentrating on the interesting special case of trees. We shall refer to the problems in this special case as the Tree Knapsack problem and the Tree Partition problem. Even these restricted problems are both NP-complete [2] and hence unlikely to be solved by "efficient" (i.e., polynomial time) optimization algorithms. However, the restriction to trees does open the door to a number of promising alternatives, in particular, pseudo-polynomial time optimization algorithms and polynomial time approximation schemes [1,2].

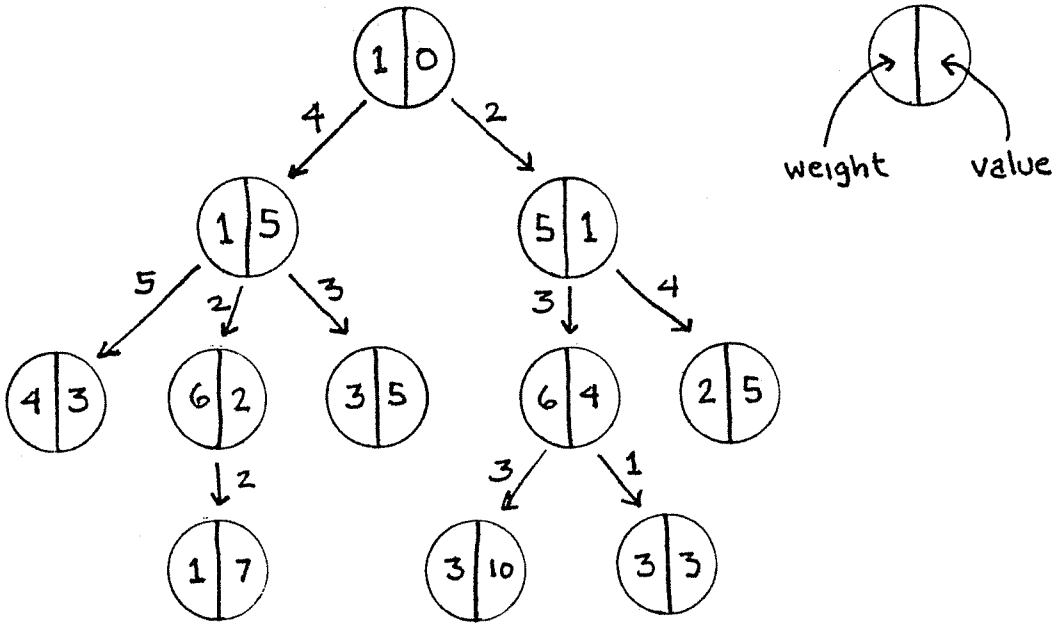


Figure 1 Sample instance of both the Partially Ordered Knapsack and Graph Partitioning problems, restricted to trees. (Directions on edges and vertex values are irrelevant when the figure is considered as a Graph Partitioning instance.) Can the reader find the optimal solution values when B = 15?

Let $n = |V|$, $W = w(V)$, and $P = p(V)$ [$P = p(E)$ in the case of Tree Partition]. The fact that these problems are NP-complete means only that they are unlikely to be solved by optimization algorithms that run in time bounded by a polynomial in n, $\log B$, $\log W$, and $\log P$. This still leaves open the possibility that they may be solved by optimization algorithms that run in pseudo-polynomial time, i.e., in time bounded by a polynomial in n, B, W, and P. Indeed, Lukes [7] presents an algorithm for Tree Partition which finds an optimal solution in time $O(nB^2)$.*

One advantage of pseudo-polynomial time algorithms is that they may turn out to be practical. Lukes' algorithm would clearly be quite efficient if B is not too large. A second advantage is that pseudo-polynomial optimization algorithms can often be turned into fully polynomial time approximation schemes [1,2]: algorithms which, given a problem instance and a proposed error bound $\varepsilon > 0$, find a feasible solution with error ratio ε or less**, and have running times bounded by a polynomial function of instance size and $1/\varepsilon$ (for instance, $O(n^2/\varepsilon)$).

Pseudo-polynomial time algorithms are usually based on dynamic programming techniques, and the standard approach to dynamic programming on trees is what we shall call the "bottom-up" approach. This is the approach taken by Lukes' algorithm for Tree Partition, and we

---

*Under the standard assumption that our computer's memory registers are big enough to hold numbers as large as P and W.

**If a solution has value S and the optimal solution value is $S^* > 0$, the error ratio is defined to be $|S-S^*|/S^*$.

observe in this paper that it is also applicable to the Tree Knapsack problem (Ibarra and Kim, in their paper on this problem [4], seem not to have noticed this fact, and hence derive approximation schemes which are exponential in $1/\varepsilon$). Our primary contribution, however, is to present a new approach to dynamic programming on trees, which we shall call "left-right" dynamic programming, and which we shall show yields improved algorithms for both the Tree Knapsack and Tree Partition problems.

In Section 2, we introduce this new approach and compare it with the bottom-up approach, using the "Out-tree" Knapsack problem as an illustration. The left-right algorithm is seen to yield a worst-case running time $\theta(nP^*)$ as opposed to $\theta(n(P^*)^2)$ for the bottom-up algorithm.*

In Section 3, we show that the left-right approach can also be applied to the "In-tree" Knapsack problem, although only the bottom-up approach seems to be general enough to handle the case of arbitrarily directed trees. In Section 4, we show that the left-right approach is not limited to knapsack-like problems by adapting it to solve the Tree Partition problem in time $\theta(n^2P)$, a running time one would normally expect to dominate the $\theta(nB^2)$ or $\theta(nP^2)$ obtainable by the bottom-up approach.

In Section 5, we consider the approximation schemes deriveable from our pseudo-polynomial time algorithms, and see that here the

---

*$f(n) = \theta(g(n))$ if $f(n) = O(g(n))$ and there is no function $h(n) = o(g(n))$ with $f(n) = O(h(n))$.

left-right algorithms all dominate their bottom-up counterparts, even when the running times of the corresponding pseudo-polynomial time algorithms are not strictly comparable.

One drawback of all these algorithms is their rather large storage requirements: as much as nP^* for the optimization algorithms. In Section 6 we show how this can be reduced to $\theta(\log n \cdot P^*)$ if we are willing to spend more time in computation.

We conclude in Section 6 with a discussion of the complexity issues that surround these problems and rule out the possibility of extending our results for trees to similar results for general graphs. In particular, we show that both the Partially Ordered Knapsack and the Graph Partitioning problems are NP-complete in the strong sense [1,2] for general graphs, and hence cannot be solved by pseudo-polynomial time optimization algorithms unless $P = NP$.

2. Out-Tree Knapsacks and the Left-Right Approach

An out-tree $T = (V,A)$ is a tree with a distinguished vertex $v_1 \in V$ called the root, such that every arc is directed away from the root. In this section we shall describe the left-right approach to dynamic programming on out-trees and show how it can be applied to the Tree Knapsack problem restricted to such trees.* For the sake of comparison, however, we first review the bottom-up approach. Dynamic programming procedures on trees generally work by solving a set of problems for each of a collection of subtrees. The solutions for a given subtree are obtained by combining those for its subtrees. The bottom-up approach can be implemented in terms of a collection of subtrees $T[v,i]$ of the given tree $T = (V,A)$ as follows:

Let $v_1,v_2,\ldots,v_n$ be a depth first ordering of the vertices of V, starting with the root. For each $v \in V$, let $d(v)$ be the number of arcs in A that are directed out of v, where $d(v) = 0$ if v is a leaf. For each $v \in V$ and i, $0 \leq i \leq d(v)$, $T[v,i]$ is the subtree of T induced by v, its first i children (in order of index), and all their successors. See Figure 2a. Note that $T[v_1,d(v_1)] = T$ and, for each $v \in V$, $T[v,0]$ is the subtree of consisting of the vector v alone.

For each subtree $T[v,i]$, the bottom-up approach will normally compute a solution $\bar{X}[v,i] = (X[v,i,0], X[v,i,1],\ldots,X[v,i,Q])$ where Q is some bound determined by the problem instance. In the Out-tree Knapsack problem, Q is taken to be some upper bound on the optimal

---

*Even this special case of Tree Knapsack is NP-complete, as can be proved by a simple transformation from the PARTITION problem [2].

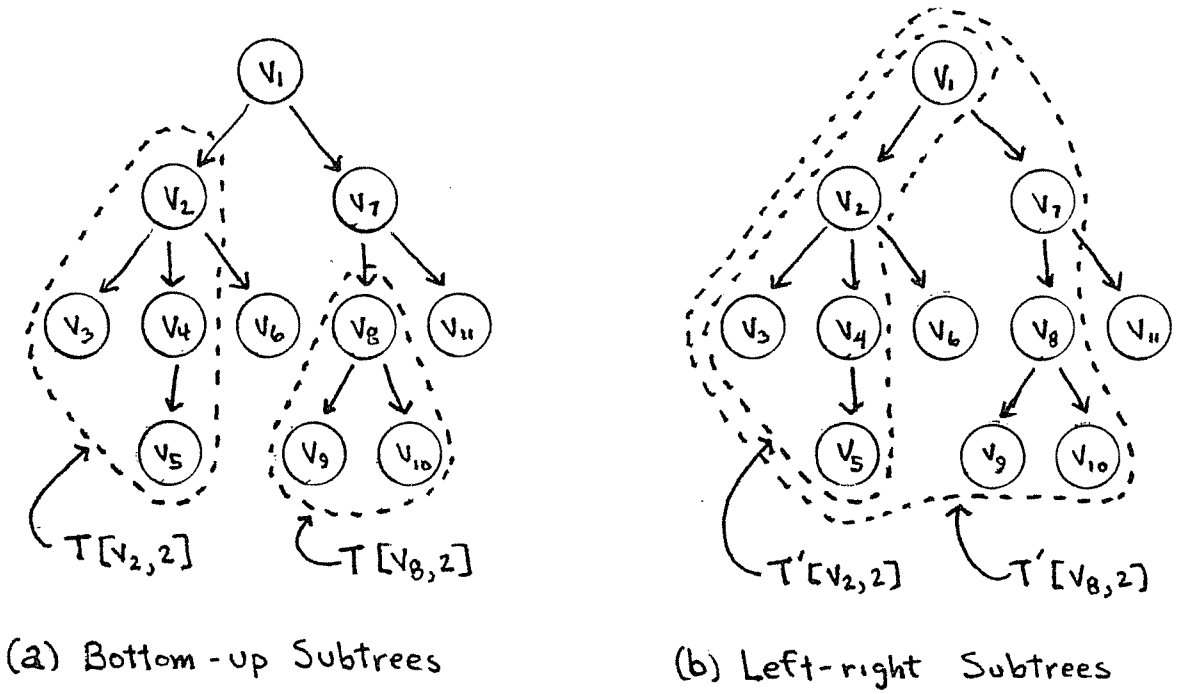


Figure 2 Examples of Subtrees used in bottom-up and left-right algorithms.

solution value, say P. The vector for this problem, which we shall denote by $\bar{X}_0[v,i]$, is defined as follows:

A solution for a subtree $T[v,i] = (V',A')$ will be a non-empty subset $V'' \subset V'$ that is closed under predecessor (with respect to $T[v,i]$) and has total weight B or less. Note that V'', being non-empty and closed under predecessor, must contain v. For each triple $[v,i,q]$, $v \in V$, $0 \leq i \leq d(v)$, $0 \leq q \leq Q$, we define

$$X_0[v,i,q] = \text{MIN}\ (\{\infty\} \cup \{w(V''):\ V''\ \text{is a solution for}\ T[v,i]\ \text{and}\ p(V'') \geq q\}).$$

The optimal solution value P^* is then seen to be

$$P^* = \text{MAX } \{q:\ 0 \le q \le Q \ \text{ and } \ X_0[v_1, d(v_1), q] < \infty\}\ .$$

The bottom-up approach gets its name from the order in which the vectors $\bar{X}[v,i]$ are computed. According to standard (if upside-down) conventions of computer science, trees are drawn, as in Figure 1, with the root on top and the leaves at the bottom. In the bottom-up approach, the values of $\bar{X}[v,d(v)]$ depends only on v and the values $\bar{X}[w,d(w)]$ for subtrees $T[w,d(w)]$ where w is a successor of v, i.e., for subtrees which are <u>below</u> v. The computation rule for the out-tree Knapsack problem is typical. For each triple $[v,i,q]$

$$(A_0) \quad \text{If } \ i = 0,\ X_0[v,i,q] = \begin{cases} w(v) & \text{if } \ q \le p(v) \\ \infty & \text{if } \ q > p(v) \end{cases}$$

$$(B_0) \quad \text{If } \ 1 \le i \le d(v) \ \text{ and } \ w \ \text{ is the } i^{th} \text{ child of } \ v$$

$$X_0[v,i,q] = \text{MIN} \left(\begin{array}{l} \{X_0(v,i-1,q]\} \cup \\ \{X_0[w,d(w),q'] + X_0[v,i-1,q-q']: \\ 0 \le q' \le q \ \text{ and the sum does not} \\ \text{exceed } \ B\} \end{array} \right)$$

That (A_0) and (B_0) correctly define $\bar{X}[v,i]$ is straightforward to verify. For (B_0), we merely observe that a solution for $T[v,i]$ either does <u>not</u> contain w, in which case it is a solution for $T[v,i-1]$, or else it does contain w, in which case it is the union of a solution for $T[v,i-1]$ and one for $T[w,d(w)]$.

The major contribution to the running time of an algorithm based on these rules is the time to carry out (B_0), which is easily seen to be $\theta(Q)$, yielding a time of $\theta(Q^2)$ for $\bar{X}[v,i]$, or $\theta(nQ^2)$ overall, since there are at least $n-1$ vectors of the form $\bar{X}[v,i]$ with $i>0$ which must be computed before we can determine P^*. The quadratic contribution of the bound Q is typical of the bottom-up approach.

The left-right approach avoids this factor, and attains a time bound of $\theta(nQ)$. The key idea is a different way of subdividing the main problem into subproblems. Given our depth-first ordering $v_1,v_2,\ldots,v_n$ of the vertices of T, let $T'[v,i]$ be the subtree of T induced by v, the first i children of v and all their successors, and all vertices in V with indices lower than that of v. See Figure 2b. We order the subtrees so that $T'[v,i]$ precedes $T'[v,i+1]$ for all $v \in V$ and i, $0 \leq i < d(v)$, and so that, if w is the i^{th} child of v, then $T'[v,i-1]$ precedes $T'[w,0]$ and $T'[w,d(w)]$ precedes $T'[v,i]$. Note that in this ordering each subtree actually contains all the subtrees that precede it. Starting with $T'[v_1,0]$, we can view the subtrees as gradually expanding: first down the left edge of the tree and then across the tree to the right. Hence the term "left-right". Note also that certain subtrees will be identical as trees ($T'[v,i]$ and $T'[w,d(w)]$ for example, where w is the i^{th} child of v), although we shall continue to consider them as separate subproblems.

We shall denote by $\bar{Y}[v,i]$ the solution vector computed for the subtree $T'[v,i]$ in the left-right approach. In the case of the Out-tree Knapsack problem, the vectors are defined as follows:

A solution for a subtree $T'[v,i] = (V', A')$ is any subset $V'' \subseteq V'$ which contains v, is closed under predecessor, and has total weight B or less. (Thus, in the above example, although $T'[v,i]$ and $T'[w,d(w)]$ are identical as trees, a solution for the second must contain w, while a solution for the first need not.) For each triple $[v,i,q]$, we define

$$Y_1[v,i,q] = \text{MIN } (\{\infty\} \cup \{w(V''): V'' \text{ is a solution for } T'(v,i) \text{ and } p(V'') \geq q\}).$$

Once again, note that $P^* = \text{MAX } \{q: 0 \leq q \leq Q \text{ and } Y_1[v_1,d(v_1),q] < \infty\}$, so long as $Q \geq P^*$.

The left-right approach computes the vectors $\bar{Y}_1[v,i]$ in the order specified above for the subtrees $T[v,i]$, according to the following easily verified rules

(A_1) If $i = 0$ and $v = v_1$, $Y_1[v,i,q] = \begin{cases} w(v) & \text{if } q \leq p(v) \\ \infty & \text{otherwise} \end{cases}$

(B_1) If $i = 0$, v is the j^{th} child of u, and $r = \text{MAX } \{0, q-p(v)\}$

$$Y_1[v,i,q] = \begin{cases} Y_1[u,j-1,r] + w(v) & \text{if the sum does not exceed } B \\ \infty & \text{otherwise} \end{cases}$$

(C_1) If $1 \leq i \leq d(v)$ and w is the i^{th} child of v

$$Y_1[v,i,q] = \text{MIN } \{Y_1[v,i-1,q], Y_1[u,d(u),q]\}.$$

Note that in no case is more than a constant amount of work required to compute any particular value of $Y_1[v,i,q]$. We thus can compute P^* in time $\theta(nQ)$ with this left-right approach, as claimed, and as opposed to the bottom-up $\theta(nQ^2)$, a quite significant speed-up if Q is large.

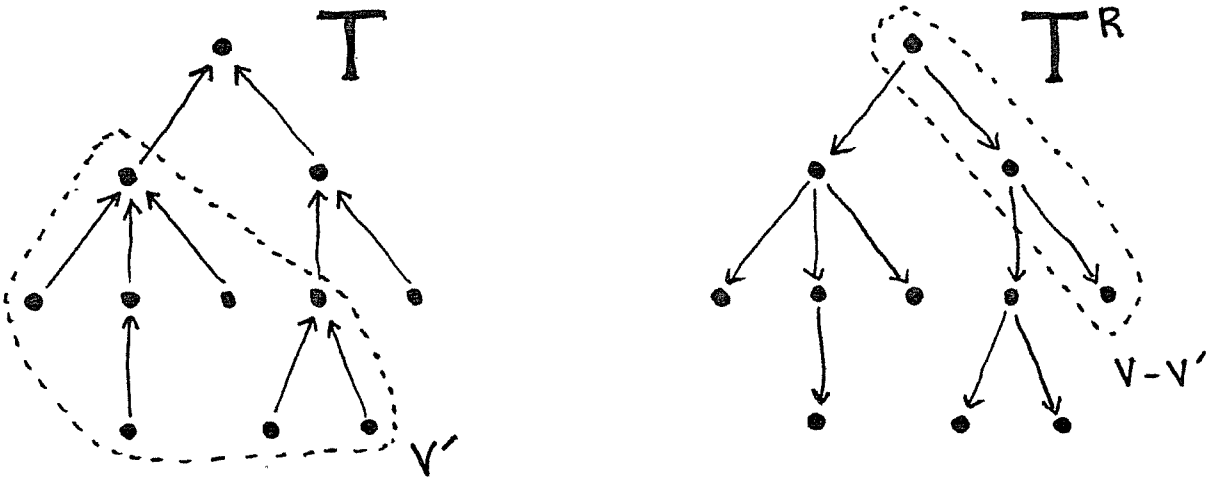
As originally specified, Q was supposed to be an upper bound on P^*, with $P = p(V)$ a suggested value. This value could be considerably larger than P^*, and so a value of Q closer to P^* would be desirable. Such a value can be obtained by using the algorithm iteratively: For any $Q \geq 0$, define $P_Q = \text{MIN}\ \{P^*,Q\}$. It is easy to see that both our algorithms actually compute P_Q, when the possibility of $Q < P^*$ is allowed. By applying the algorithm iteratively, starting with the obvious lower bound on P^* provided by $Q = \text{MAX}\ \{p(v): v \in V\}$ and doubling Q after each iteration until $P_Q < Q$, we will find P^* in time $\theta(nP^*)$ using the left-right approach ($\theta(n(P^*)^2)$ using bottom-up).

To actually construct an optimal solution, rather than merely compute its value as we have done here, requires little additional effort. One would store "unwinding variables" at the same time one was computing the solution vectors, and then "unwind" the solution in a final, linear time pass after P^* was computed. The details are more or less standard — see [3,5]. The standard techniques, however, are not very space efficient, and we shall return to this question when we discuss storage reduction techniques in Section 6.

3. In-Trees and Others

An <u>in-tree</u> $T = (V,A)$ is a tree with a distinguished root v_1, such that every arc is directed <u>toward</u> the root. Although the left-right approach seems restricted to out-trees, we can apply it to the In-tree Knapsack problem by making a simple transformation.

If $T = (V,A)$ is an in-tree, let $T^R = (V,A^R)$ denote the out-tree obtained by reversing the directions on all the arcs. See Figure 3. Note that if $V' \subseteq V$ is a Knapsack solution for T, then the set $V - V'$ has the following properties: (i) $V - V'$ is closed under predecessor with respect to T^R, (ii) $w(V-V') = W - w(V') \geq W - B$, and (iii) $p(V-V') = P - p(V')$. Thus the Tree Knapsack problem for the in-tree T corresponds to the following dual problem on T^R: Find a subset $V'' \subset V$ which is closed under predecessor, has $w(V'') \geq W - B$, and such that $p(V'')$ is minimized. This is now a problem to which the left-right approach can be directly applied.



<u>Figure 3</u> An in-tree T and its reversal T^R, with sets closed under predecessor.

Given an in-tree instance T of the Tree Knapsack problem, let subtrees $T'[v,i]$ of T^R be defined as in Section 2. A <u>solution</u> for a subtree $T'[v,i] = (V',A')$ will be a subset $V'' \subseteq V'$ which contains v and is closed under predecessor in T^R. Given a bound Q we define solution vectors $\bar{Y}_2[v,i]$ as follows. For each triple $[v,i,q]$,

$$Y_2[v,i,q] = \text{MAX } (\{-\infty\} \cup \{w(V''): V'' \text{ is a solution for } T'[v,i] \text{ and } p(V'') = q\}) .$$

Note that this definition differs from that for $\bar{Y}_1[v,i]$ in that MIN is replaced by MAX, and also that we require equality of $p(V'')$ and q. If we let

$$P_Q = \text{MIN } (\{\infty\} \cup \{q: 0 \leq q \leq Q \text{ and } Y_2[v_1,d(v_1),q] \geq W - B\})$$

then it should be clear that $P_Q = \begin{cases} \infty & \text{if } Q < P - P^* \\ P - P^* & \text{otherwise} \end{cases}$

Thus computing the values of the $\bar{Y}_2[v,i]$ will enable us to compute P^*, as long as we choose $Q \geq P - P^*$. For instance, once again $Q = P$ will do.

The values are computed in the same order as before, and the reader may readily verify that the following rules will suffice:

(A2) If $i = 0$ and $v = v_1$, $Y_2[v,i,q] = \begin{cases} w(v) & \text{if } q = p(v) \\ -\infty & \text{otherwise} \end{cases}$

(B_2) If $i = 0$ and v is the j^{th} child of vertex u

$$Y_2[v,i,q] = \begin{cases} Y_2[u,j-1,q-p(v)] + w(v) & \text{if } q \geq p(v) \\ -\infty & \text{otherwise} \end{cases}$$

(C_2) If $1 \leq i \leq d(v)$ and w is the i^{th} child of v

$$Y_2[v,i,q] = \text{MAX } \{Y_2[v,i-1,q],\ Y_2[w,d(w),q]\}\ .$$

Once again, the running time is $\theta(nQ)$ and we can use this basic procedure, suitably augmented and iterated, to find an optimal solution to the in-tree problem in time $\theta(n(P-P^*))$. In this case the comparison to the bottom-up approach will depend on the value of P^*, since the latter can still be implemented to run in time $\theta(n(P^*)^2)$. However, as we shall see in Section 5, the superiority of the left-right approach becomes clear-cut when we consider the approximation schemes derived from these algorithms.

It should be pointed out, however, that the bottom-up approach can be used to solve the Tree Knapsack problem for <u>arbitrary</u> directed trees in time $\theta(n(P^*)^2)$, whereas the left-right approach seems limited to in-trees and out-trees. In an arbitrary directed tree, there need not be any natural candidate for root, so we just pick an arbitrary vertex as v_1, and define the bottom-up subtrees $T[v,i]$ in terms of a depth-first search from this choice. A <u>solution</u> for a subtree $T[v,i] = (V',A')$ is a (possibly empty) subset $V'' \subseteq V'$ which is closed under prececessor and has $w(V'') \leq B$. Note that it need not contain v. We proceed by computing <u>two</u> solution vectors $\bar{X}_3[v,i]$ and $\bar{Z}_3[v,i]$, defined as follows:

$$X_3[v,i,q] = \text{MIN}\ (\{\infty\} \cup \{w(V''):\ V''\ \text{is a solution for}\ T[v,i] \text{ which contains } v, \text{ and } p(V'') \geq q\})$$

$$Z_3[v,i,q] = \text{MIN}\ (\{\infty\} \cup \{w(V''):\ V''\ \text{is a solution for}\ T[v,i] \text{ which does not contain } v, \text{ and } p(V'') \geq q\})$$

We then define $P_Q = \text{MAX}\ \{q: \text{MIN}\{X_3[v_1,d(v_1),q],\ Z_3[v_1,d(v_1),q]\} < \infty\}$ and observe that $P_Q = \text{MIN}\ \{Q,P^*\}$. The actual computation rules for the vectors $\bar{X}_3[v,i]$ and $\bar{Z}_3[v,i]$, which lead to the $\theta(n(P^*)^2)$ algorithm, are left as an exercise to the reader.

We conclude this section by noting that our results for the Tree Knapsack problem can be extended to the case of forests by the standard trick of combining a collection of trees into a single tree by use of a new vertex v_0 as a common root.

4. The Tree Partition Problem

In this section we show that the left-right approach can be usefully adapted to a problem that at first seems far afield from task of finding sets of vertices which are closed under predecessor in an out-tree. Recall from Section 1 that in the Tree Partition problem our goal is to find a partition $\Pi = \{V_1, V_2, \ldots, V_m\}$ of the vertices of an undirected tree T such that no set V_i has total weight exceeding B and such that $p(\Pi)$, the total value of the edges in the set $\{\{u,v\} \in E: \{u,v\} \not\subseteq V_i, 1 \leq i \leq m\}$, is minimized. Note that we may assume that each set V_i is connected (i.e., induces a connected subgraph of T), since otherwise Π can be replaced by a finer partition with the same value, merely be replacing V_i by its connected components.

The bottom-up approach to this problem, as applied by Lukes [7], results in an algorithm with running time $\theta(nB^2)$. A dual version of this (which runs in time $\theta(n(P^*)^2)$) is necessary if we are to construct approximation schemes. Our left-right algorithm will actually share some of the framework of this latter bottom-up algorithm, so let us discuss it briefly.

We define subtrees $T[v,i]$ by choosing an arbitrary vertex v_1 as root and labelling the remaining vertices by depth-first search. A <u>solution</u> for a subtree $T[v,i] = (V', E')$ is a partition $\Pi = (V_1, V_2, \ldots, V_k)$ of V' such that no set V_i has $w(V_i) > B$. Given a bound Q, we define vectors $\bar{X}_4[v,i]$ by

$$X_4[v,i,q] = \text{MIN}\ (\{\infty\} \cup \{w: \text{there is a solution } \Pi \text{ for } T[v,i] \text{ with } p(\Pi) = q \text{ and such that the set of } \Pi \text{ containing } v \text{ has weight } w\}).$$

We then have $P^* = \text{MIN}\ \{q: 0 \leq q \leq Q \text{ and } X_4[v_1, d(v_1), q] \leq B\}$, so long as $Q \geq P^*$. The reader should be able to fill in the computation rules to obtain a $\theta(n(P^*)^2)$ algorithm for finding an optimal partition.

The left-right approach is more complicated and yields a running time of $\theta(n^2P^*)$ which will nevertheless dominate the above so long as the typical situation of $P^* > n$ holds. We start with a definition based on the bottom-up approach. Let

$$P^*(u) = \text{MIN}\ \{p(\Pi): \Pi \text{ is a solution for } T(u,d(u))\}\ .$$

An optimal partition for $T[u,d(u)]$ will be one whose value equals $p^*(u)$. We shall call a solution Π for $T[u,d(u)] = (V', A')$ a normal solution if for each $v, w \in V'$ with w a child of v and v and w in different sets of Π, Π induces an optimal partition on $T[w,d(w)]$. Note that, although not all solutions need be normal, all optimal solutions must be.

In particular, an optimal partition Π for $T[u,d(u)] = (V', E')$ must consist of a set V_1 containing u, together with optimal partitions of $T[w,d(w)]$ for each $w \in V' - V_1$ which is a child of a vertex in V_1. Suppose we knew $p^*(w)$ for all vertices $w \in V'$ except u. Then the value $p^*(u)$ for an optimal partition for $T[u,d(u)]$ would be the minimum, over all connected sets $V_1 \subseteq V'$ which contain u and have total weight B or less, of

$$\sum_{v \in V_1,\ w \notin V_1,\ \{v,w\} \in E'} [p(\{v,w\}) + p^*(w)]\ .$$

Moreover, note that V_1, being connected and containing u, must be closed under predecessor with respect to the out-tree obtained from $T[u,d(u)]$ by choosing u as root and directing all edges away from it. This suggests that the left-right approach could be used to compute $p^*(u)$, given the values $P^*(w)$ for all $w \in V' - \{u\}$. (Actually, we compute quantities $P^*_Q(u)$ which equal $P^*(u)$ if $Q \geq P^*(u)$ and otherwise are infinite).

If v is a descendant of u, define the subtree $T'[u,v,i,]$ to be the intersection of $T[u,d(u)]$ and $T'[v,i]$, i.e., the subtree of $T[u,d(u)]$ consisting of v, its first i children and all their successors, and all vertices in $T[u,d(u)]$ which have indices lower than that of v. For all relevant quadruples $[u,v,i,q]$ we then define

$$Y_5[u,v,i,q] = \text{MIN } (\{\infty\} \cup \{w \leq B: \text{ There is a normal solution } \Pi \text{ for } T'[u,v,i] \text{ with } p(\Pi) = q \text{ and } v \text{ in a set of } \Pi \text{ which contains } u \text{ and has weight } w\}).$$

Note that $P^*_Q(u) = \text{MIN } (\{\infty\} \cup \{q: Y_5[u,u,d(u),q] \leq B,\ 0 \leq q \leq Q\})$.

The recurrence relations for computing the values of $\bar{Y}_5[u,v,i]$, given the values for earlier subtrees of $T[u,d(u)]$ and the values $P^*_Q(w)$ for all descendants of u, are as follows:

(A_5) If $i = 0$ and $v = u$, $Y_5[u,v,i,q] = \begin{cases} w(v) & \text{if } q = 0 \\ \infty & \text{otherwise} \end{cases}$

(B_5) If $i = 0$ and $v \neq u$ but is the j^{th} child of some vertex u'

$$Y_5[u,v,i,q] = \begin{cases} Y_5[u,u',j-1,q] + w(v) & \text{if this sum does not exceed } B \\ \infty & \text{otherwise} \end{cases}$$

(c_5) If $1 \leq i \leq d(v)$ and w is the i^{th} child of v

$$Y_5[u,v,i,q] = \begin{cases} \text{MIN } \{Y_5[u,v,i-1,q-P_Q^*(w) - P(\{v,w\})], \\ \qquad Y_5[u,w,d(w),q]\} & \text{if } q \geq P_Q^*(w) + P(\{v,w\}) \\ Y_5[u,w,d(w),q] & \text{otherwise} \end{cases}$$

Using these recurrences, we can compute $P_Q^*(u)$ in time $\theta(nQ)$. Using them repeatedly in a bottom-up fashion (this is clearly a hybrid algorithm) we can thus compute all the values $P_Q^*(u)$ for $u \in V$, and hence P_Q^*, in time $\theta(n^2Q)$. From this basic procedure we can then, by the techniques discussed in previous sections, construct an algorithm which finds optimal solutions and has running time $\theta(n^2P^*)$, as claimed.

5. Approximation Schemes

The methods for deriving fully polynomial time approximation schemes from pseudo-polynomial time optimization algorithms are illustrated in detail in [1,2,8], and can be applied in a straightforward way to the algorithms we have been discussing. The basic trick involves a "rounding" process. We start with an algorithm A_0 that finds an optimal solution if the optimal solution value P^* satisfies $P^* < Q$, and otherwise reports $P^* \geq Q$. (All our algorithms can be viewed as behaving this way.) From A_0 we construct a procedure A_1 that, given a problem instance, a desired error ratio ε, a lower bound $Q_1 \leq P^*$ and an upper bound $Q_2 \geq Q_1$, will return a solution with value P_ε satisfying $|P_\varepsilon - P^*|/P^* \leq \varepsilon$, or else report a reason for failure. In the case of our algorithms, these reasons would be

(a) $Q_2 < P^*$, bottom-up Tree Knapsack or left-right Out-tree Knapsack

(b) $Q_2 < P - P^*(1-\varepsilon)$, left-right In-tree Knapsack

(c) $Q_2 < P^*(1+\varepsilon)$, bottom-up, left-right Tree Partition

In all three cases the algorithm A_1 is obtained by constructing a modified problem instance in which values are rounded to the next multiple of $Q_1\varepsilon/n$ and then divided by $Q\varepsilon/n$, and applying A_0 to the modified instance with $Q = \frac{Q_2 n}{Q_1 \varepsilon}$. If A_0 had running time $\theta(n^\alpha Q^\beta)$, the running time for A_1 will be $\theta \frac{n^{\alpha+\beta}}{\varepsilon^\beta} (\frac{Q_2}{Q_1})^\beta)$.

To derive a true approximation scheme, we must devise ways of choosing Q_2 so that A_1 will find a solution. Assuming $\varepsilon \leq 1$, $Q_2 = P$ will suffice for Tree Knapsack and $Q_2 = 2P$ will do for Tree Partition. In order for the approximation scheme to be "fully polynomial", we need a value for Q_1 so that Q_2/Q_1 is bounded by a polynomial function of n, $1/\varepsilon$, $\log P$, $\log W$, etc. In the case of Tree Knapsack this is not difficult: $Q_1 = \max \{p(v): v \in V\}$ will suffice, since this yields $Q_2/Q_1 \leq n$. This, however, yields rather high running times: $\theta(n^5/\varepsilon^2)$ for the bottom-up approach, $\theta(n^3/\varepsilon)$ for the left-right. A considerable speed-up for both algorithms is possible if we use A_1 iteratively to find better values for Q_1 and Q_2. Moreover, such an iterative procedure seems <u>necessary</u> if we are to get bounds that make the Tree Partition scheme fully polynomial.

We illustrate this procedure in terms of our Tree Partition algorithms (the same technique carries over with minor modifications to all the other algorithms except the left-right In-tree Knapsack algorithm). Our goal is to find a Q_1 and Q_2 such that $Q_1 \leq P^* \leq Q_2$ and $Q_2/Q_1 \leq 2$. We may assume that $P^* \geq 1$, since $P^* = 0$ is trivial to detect. We perform the following loop until it halts:

1. Set $Q \leftarrow 1$
2. Perform A_1 with $\varepsilon \leftarrow 1$, $Q_1 \leftarrow Q$, $Q_2 \leftarrow 4Q$ and let P_ε be the solution value returned, if any
3. If a solution is returned, halt and return $Q_1 = \lceil P_\varepsilon/2 \rceil$, $Q_2 = P_\varepsilon$
4. Otherwise, set $Q \leftarrow 2Q$ and go to 2

First note that Q is always a lower bound on P^*. It is only doubled when no solution is returned, in which case, by (c), $4Q = Q_2 < P^*(1+\varepsilon) = 2P^*$ and hence $P^* > 2Q$. Thus algorithm A_1 is always applied correctly, and moreover will be applied at most $\lceil \log P^* \rceil$ times before it returns a solution. When this happens we must have $P_\varepsilon \geq P^*$ since this is a minimization problem, and $|P_\varepsilon - P^*|/P^* \leq \varepsilon = 1$ implies $P^* > P_\varepsilon/2$. Since P^* is an integer, we thus have $Q_1 = \lceil P_\varepsilon/2 \rceil \leq P^* \leq P_\varepsilon = Q_2$, and $Q_2/Q_1 \leq 2$, as claimed.

The total time required for iterating the loop will be $\theta(n^{\alpha+\beta} \log P^*)$, since the ε and Q_2/Q_1 contributions are here limited to a constant factor (we always have $\varepsilon = 1$ and $Q_2/Q_1 = 4$). Adding to this the time required to do a final pass with A_1, using the computed bounds $Q_1 \leq P^* \leq Q_2$ and the desired value of ε, we obtain an overall running time of $(n^{\alpha+\beta}(1/\varepsilon^\beta + \log P^*))$.

For Tree Partition, we thus obtain a left-right approximation scheme with running time $\theta(n^3(1/\varepsilon + \log P^*))$, as opposed to the bottom-up running time of $\theta(n^3(1/\varepsilon^2 + \log P^*))$: a distinct improvement despite the fact that the corresponding optimization algorithms had, strictly speaking, incomparable running times.

In the case of Out-tree Knapsack, the left-right approximation scheme devised using the above speed-up techniques has running time $\theta(n^2(1/\varepsilon + \log P^*))$ as opposed to a bottom-up running time of $\theta(n^3(1/\varepsilon^2 + \log P^*))$: a much more substantial improvement. Even in the case of In-tree Knapsack, where the speed-up techniques cannot be applied in the left-right approach because of the different nature of condition (b) from that of (a) and (c),

the original left-right time of $\theta(n^3/\varepsilon)$, obtained by using $Q_1 = \max\{p(v)\}$ and $Q_2 = P$, still beats the sped-up bottom-up time of $\theta(n^3(1/\varepsilon^2 + \log P^*))$.

Thus the value of the left-right approach is more clear-cut in the case of approximation schemes than it was in the pseudo-polynomial time optimization algorithms of previous sections. (We note in concluding this section that the "log P*" in the running time for the sped-up Tree Knapsack algorithms can be replaced by "log n", since the bound computation can in this case start with $Q = \max\{p(v)\}$ rather than $Q = 1$. This, however, will not affect the relative merits of the algorithms).

6. Reducing Storage Requirements

A drawback shared by all algorithms so far discussed is their large storage requirements. The optimization algorithms for Tree Knapsack and Tree Partition all explicitly use at least $\theta(nP^*)$ space to store the solution vectors for the various subtrees, and the approximation schemes use at least $\theta(n^2/\varepsilon)$. Since space is often the limiting factor in computations, it may well be worthwhile to try to reduce these requirements, even if this means an increase in running time. In this section we describe methods for reducing storage requirements to $\theta(\log n/n)$ times those previously specified, at a cost of increasing the running time by a factor of $n/\log n$. (Actually, these need be no increase in running time if all we want is the optimal solution value, rather than an actual solution.)

Our methods are based on similar ideas developed for code optimization in [9], although their application, at least in the case of the left-right approach, involves a certain amount of cleverness. To introduce them, let us first take a quick look at the bottom-up case. The basic bottom-up subroutine, with parameter Q, stores $2n - 1$ vectors $\bar{X}[v,i]$, each requiring space $\theta(Q)$. However, it is not really necessary to remember all of these simultaneously. For instance, once $\bar{X}[v,d(v)]$ is computed, we can forget $\bar{X}[w,i]$ for all proper descendents w of v. Hence the storage locations in which these values were stored can be re-used. In order to make possible maximum re-use of storage, however, we must properly organize our computation. This requires being more careful about our initial depth-first ordering $v_1,v_2,\ldots,v_n$ of the vertices.

For each $v \in V$, let $V[v]$ be the set of vertices consisting of v and all its successors ($V[v]$ is the vertex set for the subtree $T[v,d(v)]$). Define for each $v \in V$ the following cost function

$$C[v] = \sum_{u \in V[v]} (d(u)+1)$$

observing that this is precisely the number of subproblems that must be solved if we are to compute $\bar{X}[v,d(v)]$. We direct the depth-first ordering of V, starting at the root v_1, so that the following property is satisfied. If v_i and v_j are children of the same vertex and $i < j$, then $C[v_i] \geq C[v_j]$. The computation of solution vectors is best described recursively. Let $M[v]$ be the maximum number of solution vectors that have to be remembered to compute $\bar{X}[v,d(v)]$. We do the computation as follows:

If $d(v) = 0$, compute $\bar{X}[v,d(v)] = \bar{X}[v,0]$ and halt.
No other vectors are needed, so $M[v] = 1$. Otherwise compute $\bar{X}[u,d(u)]$ for u the first child of v, at a cost of $M[u]$, forget everything but $\bar{X}[u,d(u)]$, and then compute $\bar{X}[v,0]$ and $\bar{X}[v,1]$, after which both $\bar{X}[u,d(u)]$ and $\bar{X}[v,0]$ are forgotten. The remaining $\bar{X}[v,i]$ are computed inductively. Given $\bar{X}[v,i]$, $i < d(v)$, compute $\bar{X}[u,d(u)]$ for u the $i+1^{st}$ child of v, using $M[u]$ additional vector storage locations, then forget everything but $\bar{X}[v,i]$ and $\bar{X}[u,d(u)]$, use these to compute $\bar{X}[v,i+1]$, and then forget all but this last vector.

It is easy to see that this way of computing $\bar{X}[v,d(v)]$ yields the following recurrences:

$$M[v] = \begin{cases} 1 & \text{if } d(v) = 0 \\ \text{MAX } \{3,\ M[u_1],\ 1+M[u_j]:\ 2 \leq j \leq d(v)\} & \text{when } d(v) > 0 \\ & \text{and } u_j \text{ is the } j^{th} \text{ child of } v,\ 1 \leq j \leq d(v). \end{cases}$$

It is not difficult to see that this recurrence implies $M[v] \leq \lceil \log C[v] \rceil + 1$ since by our depth-first ordering $C[u_1] \leq \frac{1}{2}C[v]$ for all $i \geq 2$, and $C[v]$ can never equal 2 by definition. We thus can conclude that, since $C[v_1] = 2n - 1$, the total number of vector storage locations needed for calculating $\bar{X}[v_1,d(v_1)]$ is at most $\lceil \log n \rceil + 2$, for a total storage requirement of $\theta(\log n \cdot Q)$. Using the basic algorithm iteratively, we can thus find the optimal solution value P^* in storage $\theta(\log n \cdot P^*)$, in the same basic running time as before.

The corresponding technique for saving storage in the left-right approach is not as straightforward. Here, a vector $\bar{Y}[u, i-1]$ must be computed <u>before</u> we can compute $\bar{Y}[v,0]$ for v is the i^{th} child of u, and is needed again after $\bar{Y}(v,d(v)]$ has been calculated. This would make it appear that we would need to allocate enough storage to hold a number of vectors equal to the height of the tree, which for some trees is linear, not logarithmic, in n. Fortunately, there is a way around this obstacle.

Recall that, in the left-right approach, the calculation of $\bar{Y}[v,0]$, while not independent of the values of other vectors, is a very straightforward affair, given $\bar{Y}[u, i-1]$, where v is the i^{th} child of u. In fact, it is so straightforward it is reversible. Given $\bar{Y}[v,0]$ we can easily regenerate $\bar{Y}[u, i-1]$. This suggests the

following organization for our left-right computation (we assume the vertices are ordered as in the bottom-up storage reduction method).

The basic unit of computation will be viewed as that of computing $\bar{Y}[v,i]$ from $\bar{Y}[v, i-1]$ in such a way that both values are available at the end of the computation. Let $M[v,i]$ denote the maximum number of vector storage locations needed at any one time during this computation. We first compute $\bar{Y}[w,0]$ from $\bar{Y}[v, i-1]$, where w is the i^{th} child of v. If $d(w) = 0$ we then can compute $\bar{Y}[v,i]$ and $M[v,i] = 3$. Otherwise, temporarily forget $\bar{Y}[v, i-1]$ and compute $\bar{Y}[u,1]$ using the basic computation recursively. Regenerate $\bar{Y}[v, i-1]$ from $\bar{Y}[u,0]$ and forget the latter. Then compute $\bar{Y}[u,2],\ldots,\bar{Y}[u,d(u)]$ in turn, forgetting each (and all intermediate vectors) as soon as the next is desired. Finally, use $\bar{Y}[v, i-1]$ and $\bar{Y}[u,d(u)]$ to compute $\bar{Y}[v,i]$, and forget $\bar{Y}[u,d(u)]$.

It is easy to verify from this description that

$$M[v,i] = MAX \{3, M[u,1], 1 + M[u,j]: \ 2 \leq j \leq d(u)\}$$

where u is the i^{th} child of v. From this we can derive the fact that $M[v,i] \leq \lceil \log C[v] \rceil + 1$ for all i, $1 \leq i \leq d(v)$. The total number of vector storage locations needed for computing $\bar{Y}[v_1, d(v_1)]$ is thus bounded by $M[v_1,1] \leq \lceil \log n \rceil + 2$, as before for a total storage requirement of $\theta(\log n \cdot Q)$. Hence the optimal solution value can again be found using storage $\theta(\log n \cdot P^*)$ with no appreciable increase in running time over the straightforward approach.

Unfortunately, although the above techniques allow us to compute solution values using reduced storage, they do not save enough information

to enable us to construct optimal solutions once we know their values. This would normally be done by saving an auxiliary "unwinding vector" $\bar{S}[v,i]$ for each solution vector computed, whose entries would indicate how the corresponding solution vector entries were derived [3,5]. Given the optimal solution value and these vectors, an optimal solution can then be "unwound" in a single, linear-time pass. Unfortunately, storing all the vectors would require $\theta(nP^*)$ space, thus negating most of the effect of our space savings for the solution vectors.

Our answer to this problem is to store only the "last few" of the unwinding vectors $\bar{S}[v,i]$. Once the optimal value is found, we can unwind a solution as far as the remembered values go, and then recompute the rest of them as needed. If at each stage we remember $\log n$ unwinding vectors, the process may in the worst case require $n/\log n$ recomputations, and hence we will have reduced storage by a factor of $n/\log n$ at the cost of increasing running time by the same factor. Other tradeoffs are possible, and these results all carry over to the approximation schemes based on our algorithms. Details are left to the interested reader.

7. Concluding Remarks

In this paper we have considered the Partially Ordered Knapsack problem and the Graph Partitioning problem. We showed how to design pseudo-polynomial time optimization algorithms and fully polynomial time approximation schemes for these problems in the case when the underlying graph was a tree. If $P \neq NP$, this is about the best we can hope to do, since the problem is NP-complete for trees and hence cannot be solved by a polynomial time optimization algorithms unless $P = NP$. One might ask, however, whether our results for trees can be extended to the case of general graphs, perhaps be new techniques more sophisticated than bottom-up or left-right dynamic programming. Again assuming that $P \neq NP$, the answer is no.

This follows from the fact that both problems are NP-complete "in the strong sense" when general graphs are allowed as input. For a complete discussion of this concept and its implications, see [1,2]. Here it is enough to note that one of our problems will be NP-complete in the strong sense if there is a polynomial q such that the problem remains NP-complete even when restricted to instances in which no weight or value exceeds $q(n)$, where n is the number of vertices. The strong NP-completeness proofs for our problem are fairly short, so we shall sketch them here.

Recall that for NP-completeness results, we actually deal with problems stated as decision problems, rather than optimization problems. Thus the two problems under consideration become:

PARTIALLY ORDERED KNAPSACK (POK)

INSTANCE Directed acyclic graph $G = (V,A)$, a weight $w(v) \in Z_0^+$ and a value $p(v) \in Z_0^+$ for each vertex $v \in V$, a knapsack capacity $B \in Z^+$, and a bound $C \in Z^+$.

QUESTION Is there a subset $V' \subseteq V$, closed under predecessor, such that $w(V') \leq B$ and $p(V') \geq C$?

GRAPH PARTITIONING (GP)

INSTANCE Graph $G = (V,E)$, a weight $w(v) \in Z_0^+$ for each $v \in V$, a value $p(e) \in Z_0^+$ for each $e \in E$, a knapsack capacity $B \in Z^+$, and a bound $C \in Z^+$.

QUESTION Is there a partition $\Pi = (V_1,V_2,\ldots,V_m)$ of V into disjoint subsets such that $w(V_i) \leq B$, $1 \leq i \leq m$, and such that $p(\Pi) \leq C$?

Theorem 1. PARTIALLY ORDERED KNAPSACK is NP-complete in the strong sense, even if $p(v) = w(v)$ for all $v \in V$.

Proof. The proof is by a polynomial transformation from the CLIQUE problem [2]. In CLIQUE, we are given a graph $G = (V,E)$ and an integer $K \leq |V|$ and ask whether G contains a complete subgraph on K vertices. Given an instance of CLIQUE, our transformations constructs an instance I of POK as follows (we assume $|E| \geq \binom{K}{2}$, else the desired complete subgraph could not possible exist).

$$V_I = V \cup E$$

$$A_I = \{(v,e): \ v \in V, \ e \in E, \ v \text{ is an endpoint of } e\}$$

$$w_I(v) = p_I(v) = |E| + 1 \quad \text{for all} \quad v \in V$$

$$w_I(e) = p_I(e) = 1 \quad \text{for all} \quad e \in E$$

$$B_I = C_I = K(|E|+1) + \binom{K}{2} = K\left(|E| + \frac{K+1}{2}\right)$$

Note that $G_I = (V_I, A_I)$ is acyclic since all arcs are from elements of V to elements of A. Note also that all weights and values are bounded by the polynomial $q(n) = n = (|V| + |E|)$. The instance I can clearly be constructed in polynomial time. Thus all that we need show to prove that we do indeed have a polynomial transformation, and hence strong NP-completeness for POK, is that I has a solution of value C or greater if and only if G has a complete subgraph of size K.

Suppose the latter. Let (V', E') be the complete subgraph. Then $|V'| = K$ and $|E'| = \binom{K}{2} = K(K-1)/2$. The reader may readily verify that $V' \cup E'$ is the desired solution for I. Conversely, suppose I has a solution of value C or greater. Since $B_I = C_I$ and each vertex of V_I has the same value as its weight, this means that our solution $V'_I \subseteq V_I$ has value and weight exactly B. However, this can only happen if $V'_I \cap V = K$ and $V'_I \cap E = \binom{K}{2}$. Since V'_I must be closed under predecessor, this means that $(V'_I \cap V, V'_I \cap E)$ is the desired complete subgraph of G. □

Theorem 2. GRAPH PARTITIONING is NP-complete in the strong sense.

Proof. This proof is by polynomial transformation from PARTITION INTO TRIANGLES [2], in which we are given a graph $G = (V,E)$ and ask if there is a partition $\Pi = (V_1, V_2, \ldots, V_m)$ of V into disjoint subsets such that $|V_i| = 3$, $1 \leq i \leq m$ and the subgraph of G induced by V_i is a triangle, $1 \leq i \leq m$. Given an instance of this problem, our transformation constructs an instance of GP as follows. (We may assume that $|E| \geq |V|$, else the desired partition could not possibly exist.)

$$V_I = V$$

$$E_I = E$$

$$w(v) = 1 \quad \text{for all} \quad v \in V$$

$$p(e) = 1 \quad \text{for all} \quad e \in E$$

$$B = 3$$

$$C = |E| - |V|$$

Note that this transformation can clearly be performed in polynomial time, and all weights and values are bounded by the polynomial $q(n) = 1$. Thus all we need show to complete our proof is that G has a partition into triangles if and only if it has a partition into sets of size 3 or less with total value C or less. But, as the reader may readily verify, any partition meeting one of these criteria must also meet the other, so we are done. □

An interesting sidelight on these results concerns Theorem 1. Ibarra and Kim [4] have developed a polynomial (but nor fully polynomial) time approximation scheme for the PARTIALLY ORDERED KNAPSACK problem in the case when $p(v) = w(v)$ for all $v \in V$. Our result shows that no fully polynomial time approximation scheme can exist for this special case unless $P = NP$. Thus this special case is one of the few examples known of problems which cannot be solved by fully polynomial time approximation schemes (unless $P = NP$) but can be solved by polynomial time approximation schemes (another example is the INDEPENDENT SET problem for planar graphs, as can be deduced from the results in [6]).

In conclusion, we note that although our results do not extend to general graphs unless $P = NP$, there still may be hope for classes between forests and general graphs, such as the class of series-parallel graphs. There may also be more applications for our algorithmic techniques on trees. The bottom-up approach is well-known, but there may still be new worlds for it to conquer, and even in the already subjugated domains, there may be many places where a clever transformation can yield a problem to which the more efficient left-right approach is applicable. We leave the further investigation of these questions, as we have left so much of this paper, to the reader.

BIBLIOGRAPHY

1. M. R. Garey and D. S. Johnson, "Strong NP-completeness results: motivation, examples, and implications", J. Assoc. Comput. Mach. 25 (1978), 499-508.

2. M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., San Francisco, 1979.

3. O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subsets problems", J. Assoc. Comput. Mach. 22 (1975), 463-468.

4. O. H. Ibarra and C. E. Kim, "Approximation algorithms for certain scheduling problems", Math. of O.R. 3 (1978), 197-204.

5. E. L. Lawler, "Fast approximation algorithms for knapsack problems", Math. of O.R. 4 (1979), 339-356.

6. R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem", SIAM J. Comput. 9 (1980), 615-627.

7. J. A. Lukes, "Efficient algorithm for the partitioning of trees", IBM J. Res. Develop. 18 (1974), 217-224.

8. S. Sahni, "General techniques for combinatorial approximation", Operations Res. 25 (1977), 920-936.

9. R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions", J. Assoc. Comput. Mach. 17 (1970), 715-728.