Discrete Applied Mathematics 26 (1990) 159–173
North-Holland

ON AN OPTIMIZATION PROBLEM WITH NESTED CONSTRAINTS

M.E. DYER
Department of Computer Studies, University of Leeds, Leeds LS2 9JT, UK

A.M. FRIEZE
Department of Mathematics, Carnegie-Mellon University, Pittsburgh PA 15213-3890, USA; and Department of Computer Science and Statistics, Queen Mary College, London E1 4NS, UK

Received 20 June 1987
Revised 25 May 1988

We describe algorithms for solving the integer programming problem

$$\begin{aligned} &\text{maximise} && \sum_{j=1}^{n} f_j(x_j), \\ &\text{subject to} && \sum_{j \in S_i} x_j \le b_i, \quad i = 1, \ldots, m, \\ & && x_j \ge 0, \qquad j = 1, \ldots, n, \end{aligned}$$

where the f_i are concave nondecreasing and the S_i form a *nested collection* of sets. For the general problem, we present an algorithm of time-complexity $O(n \log^2 n \log \bar{b})$, where $\bar{b}$ is less than the largest of the b_i. We also examine the case in which all f_i are identical and give an algorithm requiring $O(n + m \log m)$ time. Both algorithms use only $O(n)$ space.

1. Introduction

Let $[n] = \{1, 2, \ldots, n\}$, and $\mathcal{S} = \{S_i \subseteq [n] : i \in [m]\}$. Then $\mathcal{S}$ is said to be a *nested* (or *laminar*) collection if, for $1 \le i < k \le m$,

$$S_i \cap S_k \ne \emptyset \text{ implies } S_i \subset S_k. \tag{1.1}$$

Without loss of generality below, we assume that $S_m = [n]$. This set can always be added to $\mathcal{S}$, if not already present, while retaining (1.1). For such a nested collection, the inclusion partial order may be represented by a tree. We will use the terminology of [12] with regard to trees. The tree T representing $\mathcal{S}$ will have a vertex for each $i \in [m]$ and an edge (i, k) if S_k is the (unique) smallest set properly containing S_i. The vertex m is the root of T. It is not difficult to show that any nested collection $\mathcal{S}$ has $m \le 2n - 1$. Hence T can be used to represent $\mathcal{S}$ by an $O(n)$ size data structure. Note that, if $\mathcal{S}$ is represented explicitly as a list of sets, it may require $\Omega(n^2)$ space. Since our algorithms have running times $o(n^2)$, we assume that T is already given as the representation of $\mathcal{S}$. Now, let f_j ($j \in [n]$) be nondecreasing concave functions defined on the nonnegative integers, and b_i ($i \in [m]$) be integers. The

assumptions on the f_i have the following significance. Practically, they provide a model of diminishing marginal returns to scale. Theoretically, they make the problem computationally tractable. (With no assumption on the f_i, the problem is easily shown to be NP-hard.)

Consider the following mathematical programming problem on $\mathscr{S}$:

$$\begin{aligned} &\text{maximise} && z = \sum_{j=1}^{n} f_j(x_j), \\ &\text{subject to} && \sum_{j \in S_i} x_j \leq b_i, \qquad i \in [m], \\ & && x_j \geq 0 \text{ and integer.} \end{aligned} \tag{1.2}$$

Note that we can always ensure $[n] \in \mathscr{S}$ by adding a suitable redundant constraint to (1.2). Two special cases of (1.2) are of interest. The first (1.2P) is that in which T is a simple path. The second, (1.2E), is that in which all f_j are equal, to f say. The intersection of these problems will be denoted (1.2PE).

Problem (1.2P) has been extensively studied. See, inter alia, Galperin and Waksman [8], Tamir [11] and Dyer and Walker [5]. Under an operation count model, which assumes each f_j can be evaluated in O(1) time, an $O(n \log n \log^2(b_m/n))$ time bound for (1.2P) was obtained in [5], and this was the best to date. The algorithm of [5] is based on upper bound generation within a divide-and-conquer scheme. Additionally, [5] gives a very simple $O(n)$ time algorithm for (1.2PE).

The general problem (1.2) has also been considered by Tamir [10]. He gave an $O(n^2 \log b_m)$ algorithm, generalising and improving that of [11]. As Tamir observes, (1.2P) is applicable, for example, to a simple production planning model, and then (1.2) generalises this model in a fairly natural way. Closely related problems are also studied in [1, 4].

Our results are then as follows. In Section 2 we prove various optimality conditions for (1.2). We use these in Section 3 to present a new method for these problems which uses both upper and *lower* bounds within divide-and-conquer. This not only results in a conceptually simpler algorithm for (1.2P) than that of [5], but also improves the time bound to $O(n \log n \log(b_m/n))$. The general algorithm for (1.2) then runs in time $O(n \log^2 n \log \bar{b})$, where $\bar{b}$, which satisfies $b_m/n \leq \bar{b} < b_m$, is defined in Section 3. Finally, in Section 4 we present an $O(n + m \log m)$ time algorithm for (1.2E) which generalises the $O(n)$ time algorithm of [5] for (1.2PE).

Apart from the equal functions case, we do not address the potentially interesting issue of strong polynomial complexity of algorithms for this problem. A major difficulty in trying to do this in the general case results from the fact that we choose to regard the f_i computations as being provided by an "oracle". This difficulty disappears in the equal functions case, since the exact form of f becomes irrelevant, and so, in a sense, we can "see inside" the oracle. If we were to make appropriate assumptions about the form of the f_i in the general case, it would seem we could obtain strongly polynomial algorithms, but we do not discuss this further here.

2. Optimality conditions

For notational simplicity, we will assume all singletons are in $\mathscr{S}$. First note that there is no loss in assuming that for each $i \in [m]$

$$\max_{j \in C_i} b_j \leq b_i \leq \sum_{j \in C_i} b_j, \tag{2.1}$$

where C_i is the set of children of i. This obviously implies that b_m is the largest b_i. Conditions (2.1) can be imposed in $O(n)$ time by a simple "bottom-up" preprocessing of T. We will use the following notation. For $k \in [m]$, $T(k)$ will be the subtree of T with root k and $P(k)$ the path from the root m to k. (By abuse of notation, we will make no distinction between trees and paths and their vertex sets.) We write $p(k)$ for the parent of k. Also let

$$\begin{aligned} \Delta_j(x_j) &= f_j(x_j) - f_j(x_j - 1), \quad x_j > 0, \\ &= \infty, \qquad\qquad\qquad x_j = 0. \end{aligned}$$

We assume $f_j(x_j)$ can be determined in $O(1)$ time, and hence so can $\Delta_j(x_j)$.

Now consider the Lagrangian problem for (1.2):

$$\text{maximise} \quad L = \sum_{j=1}^{n} f_j(x_j) - \sum_{i=1}^{m} \mu_i \sum_{j \in S_i} x_j. \tag{2.2}$$

This can be rewritten

$$\text{maximise} \quad L = \sum_{j=1}^{n} \left\{ f_j(x_j) - x_j \sum_{i \in P(j)} \mu_i \right\}. \tag{2.3}$$

From the theory of Lagrangian duality [6], it follows that a sufficient condition for a solution x_j^* ($j \in [n]$) to be optimal in (1.2) is that there exist μ_i^* ($i \in [m]$) satisfying

$$\mu_i^* \geq 0, \qquad i \in [m], \tag{2.4a}$$

$$\mu_i^* > 0 \text{ implies } \sum_{j \in S_i} x_j^* = b_i, \qquad i \in [m], \tag{2.4b}$$

$$\Delta_j(x_j^*) \geq \sum_{i \in P(j)} \mu_i \geq \Delta_j(x_j^* + 1), \quad j \in [n], \tag{2.4c}$$

and that x_j^* is feasible, i.e.,

$$x_j^* \geq 0, \qquad j \in [n], \tag{2.5a}$$

$$\sum_{j \in S_i} x_j^* \leq b_i, \quad i \in [m]. \tag{2.5b}$$

Suppose we write $\lambda_k = \sum_{i \in P(k)} \mu_i$ for $k \in [m]$. By convention $\lambda_{p(m)} = 0$. Then (2.4) is equivalent to the existence of λ_k^* satisfying

$$\lambda_k^* \geq \lambda_{p(k)}^*, \qquad k \in [m], \tag{2.6a}$$

$$\lambda_k^* > \lambda_{p(k)}^* \text{ implies } \sum_{j \in S_k} x_j^* = b_k, \quad k \in [m], \tag{2.6b}$$

$$\Delta_j(x_j^*) \geq \lambda_j^* \geq \Delta_j(x_j^* + 1), \qquad j \in [n]. \tag{2.6c}$$

Unfortunately, it does not follow from Lagrangian duality that we can find an *integer* solution $\underline{x}^*$ such that $\underline{\lambda}^*$, proving optimality, exists.

We will prove this by exhibiting an algorithm which constructs the appropriate $\underline{x}^*$, $\underline{\lambda}^*$. To avoid trivialities, we assume $b_i > 0$ ($i \in [m]$). If $b_i < 0$ for any i, the problem is infeasible, and, if $b_i = 0$ for any i, then S_i can be removed, setting $x_j^* = 0$ ($j \in S_i$).

```
          TREEGREEDY
1         T' := T; x_j := 0 (j ∈ [n]); s_i := b_i (i ∈ [m]); δ := ∞.
2         while T' ≠ ∅ do
            begin
3             L' := {j ∈ [n]; j is a leaf of T'};
4             δ := Δ_r(x_r + 1) = max_{j ∈ L'} Δ_j(x_j + 1)
              (in case of ties choose smallest such r);
5             x_r := x_r + 1;
6             s_k := s_k − 1 (k ∈ P(r));
7             if s_k = 0 for any k ∈ P(r) then
                begin
8                 i := vertex on P(r) nearest root m such that s_i = 0;
9                 λ*_k := δ (k ∈ T'(i)); x*_j := x_j (j ∈ L' ∩ T'(i));
10                T' := T' − T'(i)
11              end
12          end
```

Since all the f_j are concave, the value of δ generated at line 3 of TREEGREEDY is nonincreasing. It is then not difficult to verify that the values assigned in line 9 satisfy conditions (2.5), (2.6). ($\underline{x}^*$ is feasible since $\sum_{j \in S_i} x_j \leq b_i$ throughout. (2.6a) follows from the fact that δ decreases, $\lambda_k^* > \lambda_{p(k)}^*$ means that k was the root of a tree T' removed in line 9 and so (2.6b) follows from $s_k = 0$ here. Line 4 implies that $\delta \geq \Delta_j(x_j + 1)$ for $j \in L'$ which implies $\lambda_j^* \geq \Delta_j(x_j^* + 1)$. Finally if $\lambda_j^* > \Delta_j(x_j^*)$, then $x_j^* > 0$ and so at some stage $j \in L'$ and $\delta = \Delta((x_j^* - 1) + 1)$. But δ cannot then increase to λ_j^*.) In fact, with the rule for resolving ties in line 4, TREEGREEDY produces the lexicographically largest optimal solution. A fairly obvious implementation of TREEGREEDY will run in $O(nb_m)$ time. Less obviously, by storing the $\Delta_j(x_j + 1)$ ($j \in L'$) values in a priority queue, and using the data structure of Sleator and Tarjan [9], [12] for manipulating trees, it can be implemented in $O(b_m \log n)$ time. For $b_m = O(n)$, this will have time bound $O(n \log n)$, which is better than the bound for our algorithm CONQUERTREE below. Thus we will assume that $b_m/n \to \infty$ with n below. Although TREEGREEDY has some computational value, we will use it chiefly to establish properties of the optimal solution which we can then use to develop a more refined algorithm. In what follows, when we use the term "optimal solution", we will always mean the optimal solution produced by TREEGREEDY.

The fact that a "greedy" algorithm solves (1.2) also follows from the general theory of polymatroids. See, for example, the articles by Lovász and Schrijver in the book [2]. However, it is more convenient here to give a self-contained development.

For a given $i \in [m]$, let $\Pi(i)$ be the problem which contains only the constraints and variables in $T(i)$, i.e.,

$$\begin{aligned} &\text{maximise} && \sum_{j \in S_i} f_j(x_j), \\ &\text{subject to} && \sum_{j \in S_k} x_j \leq b_k, \quad k \in T(i), \\ & && x_j \geq 0. \end{aligned}$$

Thus $\Pi(m)$ is (1.2). Let $x_j^{(i)}$ be the values of the variables in the optimal solution to $\Pi(i)$.

Lemma 2.1. *$x_j^{(m)} \leq x_j^{(i)}$ for all $j \in S_i$.*

Proof. The progress of TREEGREEDY in solving $\Pi(m)$ and $\Pi(i)$ is identical (in $T(i)$) at least up to the point where $T(i)$ is removed, possibly as part of a larger tree in line 10 when solving $\Pi(m)$. In $\Pi(m)$ all variables in $T(i)$ are now fixed, whereas in $\Pi(i)$ they may possibly be increased further. □

(This lemma can be strengthened somewhat, and a strengthening may have computational significance, as noted by a referee, but we will not pursue this here.) Lemma 2.1 shows that, by solving problems on subtrees, we can impose upper bounds on variables. More importantly:

Lemma 2.2. *The upper bounds $x_j \leq x_j^{(i)}$ $(j \in S_i)$ imply the constraints $\sum_{j \in S_k} x_j \leq b_k$ $(k \in T(i))$.*

Proof. $\sum_{j \in S_k} x_j \leq \sum_{j \in S_k} x_j^{(i)} \leq b_k$. □

Thus we can replace subtree constraints by simple upper bounds after solving $\Pi(i)$. This merely generalises a result of [5]. The following idea, however, is new. For $i \in [m]$, let $\bar{\Pi}(i)$ be the problem obtained in $T - T(i)$ by assuming $\sum_{j \in S_i} x_j = b_i$, i.e., where $\bar{S}_k = S_k - S_i$ $(k \in P(i))$,

$$\begin{aligned} &\text{maximise} && \sum_{j \in \bar{S}_m} f_j(x_j), \\ &\text{subject to} && \sum_{j \in \bar{S}_k} x_j \leq b_k - b_i, \quad k \in P(i), \\ & && \sum_{j \in S_k} x_j \leq b_k, \qquad k \notin T(i) \cup P(i), \\ & && x_j \geq 0. \end{aligned}$$

Let $\bar{x}_j^{(i)}$ be the optimal solution to $\bar{\Pi}(i)$.

Lemma 2.3. *$x_j^{(m)} \geq \bar{x}_j^{(i)}$ for all $j \in [n] - S_i$.*

Before proving Lemma 2.3 we require a preliminary result. Let Π denote (1.2) with optimal solution ξ_j, and Π' be the problem obtained by increasing all b_k ($k \in P(i)$) to $b'_k = b_k + 1$. Then:

Lemma 2.4. *The optimal solution ξ' to Π' satisfies $\xi'_j \geq \xi_j$ ($j \in [n]$).*

Proof. Primed quantities refer to Π'. Consider TREEGREEDY's progress in Π and Π'. They proceed identically either to termination, in which case the solutions are identical, or until $s_t = 0$ for some $t \in P(i)$ in Π. We take t as the nearest such vertex to m. At this point $s'_t = 1$ and $s'_k > 1$ for k nearer to m on $P(i)$. Now $T(t)$ will be removed in Π. The two algorithms again proceed identically either to termination or until we increase some x_r ($r \in S_t$) in line 5 in Π'. At this point $\xi'_r := \xi_r + 1$ and $T(t)$ is removed in Π'. The algorithm then proceeds identically in Π, Π' to termination. Thus ξ'_j, ξ_j are equal, except possibly $\xi'_r = \xi_r + 1$ for some r. This conclusion is stronger than the lemma. □

It should be observed that Lemma 2.4 is false if not all b_k on $P(i)$ are increased.

Proof of Lemma 2.3. Suppose $T(i)$ is deleted, when solving $\Pi(m)$ by TREEGREEDY, when $s_i = \alpha$. It follows that the optimal values $x_j^{(m)}$ ($j \notin S_i$) are given by solving the problem Π'' obtained by replacing b_k by $b''_k = b_k - b_i + \alpha$ ($k \in P(i)$) and deleting $T(i)$. But Π'' is the problem obtained from $\bar{\Pi}(i)$ by increasing all right-hand sides $\bar{b}_k = b_k - b_i$ by α for $k \in P(i)$. The conclusion now follows by applying Lemma 2.4 α times. □

Thus, using Lemma 2.3, we can impose lower bounds on variables by solving $\bar{\Pi}(i)$. We also have:

Lemma 2.5. *The lower bounds $x_j \geq \bar{x}_j^{(i)}$ together with $\sum_{j=1}^n x_j \leq b_m$ imply the constraints $\sum_{j \in S_k} x_j \leq b_k$ ($k \in P(i)$).*

Proof. First note that we may assume $\sum_{j=1}^n x_j^{(m)} = b_m$, since $T' = \emptyset$ on termination of TREEGREEDY and we are only concerned with optimal solutions that might be constructed by this method.

We may also assume $\sum_{j \in \bar{S}_m} \bar{x}_j^{(i)} = b_m - b_i$, since $\bar{\Pi}(i)$ satisfies the right-hand inequality of (2.1). Now, for $k \in P(i)$, we have

$$\sum_{j \in S_k} x_j = \sum_{j \in \bar{S}_k} x_j + \sum_{j \in S_i} x_j = \sum_{j \in \bar{S}_k} \bar{x}_j^{(i)} + \sum_{j \in \bar{S}_k} (x_j - \bar{x}_j^{(i)}) + \sum_{j \in S_i} x_j$$

$$\leq (b_k - b_i) + \sum_{j \in \bar{S}_m} (x_j - \bar{x}_j^{(i)}) + \sum_{j \in S_i} x_j$$

$$= (b_k - b_i) + \sum_{j=1}^{n} x_j - (b_m - b_i) = b_k + \sum_{j=1}^{n} x_j - b_m$$

$$\leq b_k. \qquad \square$$

Thus, using Lemmas 2.1 and 2.3 to construct upper and lower bounds, we can, by Lemma 2.5, reduce the problem to a one-constraint problem. This is the essence of our algorithm in Section 3.

3. Algorithm for the general problem

We now describe the algorithm for (1.2). We assume here that T is a data structure having the values b_i and n_i, the number of leaves in $T(i)$, for each $i \in T$. Note that all n_i can be determined in $O(n)$ total time by a simple "bottom-up" algorithm. For each leaf $j \in T$ we also have a function f_j and an output value x_j. Since the procedure is recursive, it is necessary to specify at times to which tree certain values belong. We show this by appending the tree in brackets. Thus $b_j(T')$ refers to b_j values in tree T'. Where no tree is specified, it will always be T.

```
        CONQUERTREE (T)
1       r := root of T; s := n_r;
2       if s = 1 then x_r := b_r;
        else
          begin
3           L := {leaves of T};
            i := r;
4           while i has a child k with n_k > ½n_r do i := k;
5           K := {k ∉ P(i): p(k) ∈ P(i)};
6           for k ∈ K do T' := T(k); CONQUERTREE (T');
7             for each leaf j ∈ T' do b_j := x_j(T')
8           T' := P(i);
9           for k ∈ T' do b_k(T') := b_k − b_i; n_k(T') := n_k − n_i
10          for k ∈ K, p(k) ≠ i do
11            for each leaf j ∈ T(k) do T' := T ∪ {(j, p(k))}
                b_j(T') := b_j; n_j(T') := 1.
12          CONQUERTREE (T')
13          for j ∈ L do u_j := b_j
14          if j ∈ T(i) then l_j := 0 else l_j := x_j(T')
15          SINGLE(L, l, u, b_r, y*); (see below)
16          for j ∈ L do x_j := y_j*
17        end
18      return
```

The procedure SINGLE $(L, \underline{l}, \underline{u}, b_r, \underline{y}^*)$ solves the problem

$$\text{maximise } \sum_{j \in L} f_j(y_j),$$

$$\text{subject to } \sum_{j \in L} y_j \leq b_r,$$

$$l_j \leq y_j \leq u_j, \quad j \in L,$$

by the algorithm of Frederickson and Johnson [7], to give optimal values y_j^* $(j \in L)$. Since, in the algorithm $|L| = n_r$, this takes time $O(n_r \log(b_r/n_r))$.

The rationale of CONQUERTREE is as follows. Line 2 provides the obvious solution to a one-variable problem. Otherwise line 4 constructs a path $P(i)$ which decomposes T into subtrees, none of which has more than $\frac{1}{2}n_r$ leaves. Then we recursively apply the algorithm on each of these subtrees to provide upper bounds as in Lemmas 2.1, 2.2. We use these upper bounds to simplify $T - T(i)$ in lines 8–11. Then the algorithm is called recursively to provide lower bounds, as in Lemmas 2.3, 2.5. Finally, we solve a one-constraint problem, with the bounds generated, to give the optimal values for the overall problem. The validity of this procedure follows directly from the results of Section 2.

It remains to determine the time-complexity of CONQUERTREE. We do this in two stages. Let us say a tree T is a *cactus* if every vertex of T has at most one interior child. (Thus a cactus is a simple path plus single-edge "spikes".) First note that if CONQUERTREE is invoked with T a cactus, then every recursive call will also involve a cactus. The restructuring operation in step 11 generates a cactus, and is redundant if T is already a cactus. The important point, however, is that, if T is a cactus, at most one of the subtrees $T(k)$, $k \in K$, used in line 6 will have more than one vertex. Hence all but one of the calls to CONQUERTREE in line 6 will take O(1) time. Thus, let $C(n, b_m)$ be the time bound for CONQUERTREE applied to a cactus T with n leaves and root m. Then lines 1–11 occupy at most $C(\frac{1}{2}n, b_i) + O(n)$ time and line 12 at most $C(\frac{1}{2}n, b_m - b_i)$ time. Lines 13–18 then take time $O(n \log(b_m/n))$, assuming $b_m/n \to \infty$. Thus we have

$$C(n, b_m) \leq C(\tfrac{1}{2}n, b_i) + C(\tfrac{1}{2}, b_m - b_i) + An \log(b_m/n) \tag{3.1}$$

for some constant A.

Lemma 3.1. $C(n, b_m) = O(n \log n \log(b_m/n))$.

Proof. Clearly $C(1, b_m) \leq A$ if A is chosen large enough. Assume inductively that the lemma is true for all (n', b_m') lexicographically smaller than (n, b_m). Then, by (3.1)

$$C(n, b_m) \leq (\tfrac{1}{2}A)n \log(\tfrac{1}{2}n)\log(2b_i/n) + (\tfrac{1}{2}A)n \log(\tfrac{1}{2}n)\log(2(b_m - b_i)/n)$$
$$+ An \log(b_m/n)$$

$$= An\log(\tfrac{1}{2}n)\left(\tfrac{1}{2}\log\left(\frac{2b_i}{n}\right)+\tfrac{1}{2}\log\left(\frac{2(b_m-b_i)}{n}\right)\right)+An\log(b_m/n)$$

$$\le An\log(\tfrac{1}{2}n)\log(b_m/n)+An\log(b_m/n)$$

using the concavity of the log function

$$= An\log n\log(b_m/n). \qquad \square$$

This improves the time bound of [5] by a factor $\log(b_m/n)$. Note that it is at least as good, even for small b_m, as the bound given by TREEGREEDY. It differs only by a $\log n$ factor from the only known lower bound of $\Omega(n\log(b_m/n))$, provided for the one-constraint problem in [7]. We now consider the case where T is a general tree. For such a tree let I be any set of incomparable interior vertices, i.e., if $i,k\in I$, then $S_i\cap S_k=\emptyset$. Let $\mathcal{I}$ be the set of such I, and let

$$B=\max_{I\in\mathcal{I}}\sum_{i\in I}b_i, \qquad \bar{b}=B/n. \tag{3.2}$$

B can be determined in $O(n)$ time as follows. Strip the leaves from T. Successively remove each leaf (of the reduced tree) which is the only child of its parent. Then I is the set of leaves of the final tree. The validity of this procedure follows easily from (2.1). Clearly, if T is a cactus, then $B=b_m$ and $\bar{b}=b_m/n$. Also, if T contains disjoint subtrees $T(k)$, $k\in K$, with B-values B_k, then $B(T)$ satisfies

$$B\ge\sum_{k\in K}B_k. \tag{3.3}$$

This follows directly from (3.2). The inequalities

$$b_m/n\le\bar{b}\le\sum_{j=1}^{n}b_j/n\le b_m \tag{3.4}$$

also follow, from (2.1).

Now let $G(n,\bar{b})$ be the time complexity on a tree T with n leaves and $\bar{b}$ as defined by (3.2). Then lines 1–7 of CONQUERTREE take time

$$\sum_{k\in K}G(n_k,\bar{b}_k)+O(n), \tag{3.5}$$

where $n_k\le\frac{1}{2}n$, $n=\sum_{k\in K}n_k$ and $\bar{b}_k=B_k/n_k$ with B_k as in (3.3).

Now lines 8–13 construct a cactus T' and apply CONQUERTREE to this. Thus they take time at most

$$C(\tfrac{1}{2}n,b_m-b_i)=O(n\log n\log(b_m/n))=O(n\log n\log\bar{b}). \tag{3.6}$$

Finally, lines 14–18 take time $O(n\log(b_m/n))$ which is dominated by (3.6). Thus from (3.5), (3.6)

$$G(n,\bar{b})\le\sum_{k\in K}G(n_k,\bar{b}_k)+O(n\log n\log\bar{b}). \tag{3.7}$$

Lemma 3.2. $G(n,\bar{b})=O(n\log^2 n\log\bar{b})$.

Proof. Let A be the implied constant in (3.7), clearly $G(1, \bar{b}) \leq A$ for large enough A. Then, as in Lemma 3.1,

$$\begin{aligned} G(n, \bar{b}) &\leq \sum_{k \in K} A n_k \log^2 n_k \log \bar{b}_k + An \log n \log \bar{b} \\ &\leq A \log^2(\tfrac{1}{2}n) \sum_{k \in K} n_k \log \bar{b}_k + An \log n \log \bar{b} \\ &\qquad \text{since all } n_k \leq \tfrac{1}{2}n \\ &\leq A \log^2(\tfrac{1}{2}n) n \log\left(\sum_{k \in K} B_k / n \right) + An \log n \log \bar{b} \\ &\qquad \text{using the concavity of log} \\ &\leq An \log^2(\tfrac{1}{2}n) \log \bar{b} + An \log n \log \bar{b} \\ &\qquad \text{using (3.3)} \\ &\leq An \log^2 n \log \bar{b}. \qquad \square \end{aligned}$$

We can replace $\bar{b}$ by simpler estimates using (3.4). We speculate that the bound in Lemma 3.2 is probably improvable to $O(n \log n \log \bar{b})$. This is prompted by the following considerations. This bound is correct for a cactus, by Lemma 3.1. At the other extreme, if T is "balanced" (i.e., if $\exists \alpha > 0$ such that if $r \in T$, then for any child k of r, $T(k)$ has at least αn_r leaves where $T(r)$ has n_r leaves), then we can again prove a bound $O(n \log n \log \bar{b})$. This follows since the path $P(i)$ in CONQUERTREE then has length $O(1)$. Since this bound is correct at both extremes, we conjecture it is true in the general case. Note that our complexity bounds do not take account of tree structure in any direct way, and the key to improving them may be to do this.

4. Algorithm for equal functions

When all f_j are equal to f, there is a considerable simplification. Now all the $\Delta_j = \Delta$, a nondecreasing function. In place of (2.6), it becomes sufficient to determine real values y_k^* such that

$$y_k^* \leq y_{p(k)}^*, \qquad k \in [m], \tag{4.1a}$$

$$y_k^* < y_{p(k)}^* \text{ implies } \sum_{j \in S_k} x_j^* = b_k, \quad k \in [m], \tag{4.1b}$$

$$x_j^* = \lfloor y_j^* \rfloor \text{ or } \lceil y_j^* \rceil, \qquad j \in [n]. \tag{4.1c}$$

For, if we have such values, $\lambda_k^* = \Delta(\lfloor y_k^* \rfloor + 1)$, $k \in [m]$, satisfies (2.6). Since (4.1) does not involve f, the algorithm only requires the constraints as input. Of course, we cannot immediately assert that values y_k^* exist satisfying (4.1), and again our proof will be algorithmic.

If we were to solve this problem by CONQUERTREE, the one-constraint problem

$$\begin{aligned} &\text{maximise} && \sum_{j\in L} f(x_j), && (4.2)\\ &\text{subject to} && \sum_{j\in L} x_j \le b_r, \\ & && l_j \le x_j \le u_j, \quad j\in L, \end{aligned}$$

solved in line 15, can now be done in $O(n)$ time. The algorithm involves successively eliminating bounds and variables from (4.2), using a linear-time median find algorithm [3]. We will omit details, however, since they are a little complicated, and the resulting complexity of CONQUERTREE is only $O(n \log^2 n)$. Instead we will exhibit a somewhat simpler algorithm which runs in time $O(n + m \log m)$ on (1.2E). Note that here we will not require that $\mathscr{S}$ contains all (or indeed any) singletons, and thus $m = o(n)$ is possible. We do however retain the assumption $[n] \in \mathscr{S}$ since this has no influence on the complexity of the algorithm. We assume that for each $i \in [m]$ we are given as data the list

$$L_i = \{ j \in S_i : j \notin S_k \text{ for any } S_k \subseteq S_i \}.$$

Note that $\sum_{i=1}^{m} |L_i| = n$. Observe that if $j \in L_i$, then (4.1c) should now read

$$x_j^* = \lfloor y_i^* \rfloor \quad \text{or} \quad \lceil y_i^* \rceil. \tag{4.1c)'}$$

We will call these conditions (4.1)′.

We will present the algorithm in two phases. The first, YCALC, calculates the y_k^* so that they would satisfy (4.1)′ if the integrality contraints on the x_j were relaxed and (4.1c)′ is replaced by

$$x_j^* = y_i^*. \tag{4.1c)''}$$

We will call these conditions (4.1)″, and the x_j^*-values satisfying them will be denoted by $\bar{x}_j$. The second phase then rounds the $\bar{x}_j$ up or down.

For each $i \in [m]$ we have the list L_i and $l_i = |L_i|$. In addition we store the following information, requiring $O(m)$ space.

(i) $d_i = \sum_{k \in D_i} l_k$, where D_i, which is not stored, is a subtree of $T(i)$. D_i is either empty or has root i, and is such that, if $j \in L_k$, $k \in D_i$, then $\bar{x}_j = y_i$. The values $\bar{x}_j$ defined in this way are changed in value during the algorithm by the nonexecutable statements as given between *'s in line 12 of YCALC. They are not actually computed here. If D_i is nonempty, then constraint i is satisfied with equality (i.e., is "binding"). The progress of these sets D_i is recorded by commands between *'s below. Again they are not executed by the algorithm.

(ii) H_i, a meldable heap [12] of vertices k with key values y_k. The H_i are disjoint for different i, and thus $\sum_{i=1}^{m} |H_i| \le m$. Furthermore $k \in H_i$ implies that currently $D_k \neq \emptyset$. We assume that a heap of size s can be constructed in $O(s)$ time. Also,

finding and deletion of the maximum, and merging, take $O(\log m)$ time. H_i contains the vertices lying immediately "below" D_i in $T(i)$.

(iii) $h_i = b_i - \sum_{k \in H} b_k$ $(= \sum_{j \in LD_i} \bar{x}_j$ where $LD_i = \bigcup_{r \in D_i} L_r)$.

(iv) $y_i = h_i / d_i$.

```
        YCALC
1       for i := 1 to m do
2         d_i := l_i; binding (i) := true   *D_i := L_i*
3         construct H_i from the children of i;
4         h_i := b_i − Σ_{k∈H_i} b_k;
5         y_i := h_i/d_i (= sign(h_i) × ∞ if d_i = 0)
6         while y_i < maxkey(H_i) do
7           begin p := vertex with maxkey(H_i);
8             delete p from H_i; binding(p) := false
9             H_i := merge(H_i, H_p)   (destroying H_p);
10            d_i := d_i + d_p   *D_i := D_i ∪ D_p*
11            h_i := h_i + h_p;
12            y_i := h_i/d_i   *x̄_j := y_i, j ∈ LD_i*
13          end
14      for i := m to 1 do
15        if not binding(i) then y_i := y_{p(i)}
```

The time complexity of YCALC is $O(m \log m)$. First note that steps 1–5 and 14, 15 take only $O(m)$ time. For a given "stage" i at line 6, the while-loop will take $O((m_i + 1)\log m)$ time, where m_i is the number of merges at stage i. But since no vertex can play the role of p in more than one merge, $\sum_{i=1}^{m} m_i \leq m$ and the bound follows.

Lemma 4.1. *The values* y_i $(i \in [m])$ *and* $\bar{x}_j = y_i$ $(j \in L_i)$ *satisfy the conditions* (2.5), (4.1)″.

Proof. We show first that

$$\sum_{j \in S_i} \bar{x}_j = b_i \quad \text{so long as binding}(i) = \textbf{true}. \tag{4.3}$$

We prove this by induction on i. Note that $k \in H_i$ implies binding$(k) =$ **true**. Also, throughout the execution of stage i,

$$\begin{aligned}\sum_{j \in S_i} \bar{x}_j &= \sum_{j \in LD_i} \bar{x}_j + \sum_{k \in H_i} \sum_{j \in S_k} \bar{x}_j \\ &= d_i y_i + \sum_{k \in H_i} b_k \\ &\qquad \text{using the induction hypothesis} \\ &= b_i.\end{aligned}$$

We remark next that the values y_i^{new} computed in line 12 satisfy $y_i^{\text{old}} < y_i^{\text{new}} < y_p^{\text{old}}$. Hence the $\bar{x}_j$ values do not increase for $j \in L_k$ once $i > k$. It follows then from (4.3) that on termination we have

$$\sum_{j \in S_i} \bar{x}_j \leq b_i, \quad i \in [m].$$

The y_i values computed in lines 14, 15 do not decrease as we go from child to parent because lines 7–12 ensure that, on completion of stage i, $y_i \geq y_k$ for all $k \in T(i)$ with binding(k) = **true**.

We must now show how to round the $\bar{x}_j$ to integer values x_j^* so as to comply with (2.5), (4.1)′. At the end of YCALC we have values $d_i = |D_i|$ for those i with binding(i) = **true**. If binding(k) = **false**, $k \in D_i$, we need to compute $d_k = |D_i \cap T(k)|$.

```
DCALC
for i := 1 to m do
  d_i := l_i;
  for each child k of i with binding(k) = false do d_i := d_i + d_k
```

DCALC obviously takes $O(m)$ time and correctly computes the required values. Our proof will use, but we will not compute, the values

$$h_k = b_k - \sum_{r \in T(k) \cap H_i} b_r.$$

Consider a *fixed* i such that binding(i) = **true**. For $k \in D_i$ define

$$S_k' = S_k - \bigcup_{r \in H_i} S_r.$$

Note that $d_k = |S_k'|$ and that $j \in S_k'$ implies $\bar{x}_j = y_i$. We determine 0, 1 variables δ_j $(j \in S_i')$ such that

$$\delta_j = \begin{cases} 0, & \text{if } x_j^* = \lfloor y_i \rfloor, \\ 1, & \text{if } x_j^* = \lceil y_i \rceil. \end{cases}$$

It will be sufficient to choose the δ_j to satisfy

$$\sum_{j \in S_k'} \delta_j \leq h_k - d_k \lfloor y_i \rfloor, \quad k \in D_i \quad \text{(with equality for } k = i).$$

Let $f_i = y_i - \lfloor y_i \rfloor$ $(j \in S_i')$. Then $h_k - d_k \lfloor y_i \rfloor \geq \lceil d_k f_i \rceil$ and so it is sufficient to determine a solution to

$$\sum_{j \in S_k'} \delta_j \leq \lceil d_k f_i \rceil, \quad k \in D_i \quad \text{(with equality for } k = i). \tag{4.4}$$

(Note that $d_i f_i$ is integral.)

The following algorithm constructs a solution to (4.4):

```
        ROUND(i, q, f)
1       if l_i > q then r_i := q; q := 0 else r_i := l_i; q := q - l_i
2       for each child k of i with binding(k) = false do
          begin
3           w := ⌈d_k f⌉
4           if q > w then ROUND(k, w, f); q := q - w
5           else ROUND(k, q, f); q := 0
6         end
7       return
```

The procedure ROUND$(i, d_i f_i, f_i)$ calculates r_k $(k \in D_i)$ such that if we set r_k variables x_j $(j \in L_k)$ to $\lceil y_i \rceil$ and the remainder to $\lfloor y_i \rfloor$, we will obtain a solution to (4.4) with equality for $k = i$. We start at the root of D_i with "quota" of $d_i f_i$. We fill as much as possible from L_i in line 1. Otherwise we give each child k of i in D_i a quota not exceeding $\lceil d_k f_i \rceil$ in lines 3–6. Since

$$d_j = l_j + \sum_{k \in CD_j} d_k, \quad j \in D_i,$$

where CD_j are the children of j in D_i, we have

$$\lceil d_j f_i \rceil \leq l_j + \sum_{k \in CD_j} \lceil d_k f_i \rceil, \quad j \in D_i.$$

Thus the sum of the quotas we allocate to the children can always be sufficient. The time complexity of ROUND$(i, d_i f_i)$ is clearly $O(|D_i| + |H_i|)$. The $|H_i|$ term arises only from the test in line 2. Thus the following routine takes $O(m)$ time in total.

```
ROUNDTREE
for i := 1 to m do
  if binding(i) = true then f_i := y_i - ⌊y_i⌋; ROUND(i, d_i f_i, f_i)
```

Note that up to this point we only require $O(m \log m)$ time in total. We now calculate the solution in a further $O(n)$ time, giving the claimed time bound of $O(n + m \log m)$.

```
SOLUTION
for i := 1 to m do
  for j ∈ L_i do
    if r_i > 0 then x_j := ⌈y_i⌉; r_i := r_i - 1
    else x_j := ⌊y_i⌋
```

The overall algorithm consists of YCALC, DCALC, ROUNDTREE and SOLUTION in that order.

It may be noted that if T is a path of length m, then the heaps H_i in YCALC never have more than one element and the overall algorithm has time complexity $O(n)$.

In fact, the algorithm reduces essentially to that of Dyer and Walker [5] for (1.2PE). The rounding operation needed in (1.2PE) is also trivial, so the algorithm is considerably simpler than that given here. See [5] for details.

References

[1] R.D. Armstrong, P. Sinha and A.A. Zoltners, The multiple-choice nested knapsack model, Management Sci. 28 (1982) 34–43.

[2] A. Bachem, M. Grötschel and B. Korte, eds., Mathematical Programming: The State of the Art (Springer, Berlin, 1983).

[3] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest and R.E. Tarjan, Time bounds for selection, J. Comput. System. Sci. 7 (1973) 448–461.

[4] P. Brucker, Network flows in trees and knapsack problems with nested constraints, in: Proceedings of the Eighth Conference on Graph-theoretic Concepts in Computer Science, H.J. Schneider and H. Göttler, eds. (Hanser, Munich, 1982) 25–35.

[5] M.E. Dyer and J. Walker, An algorithm for a separable integer programming problem with cumulatively bounded variables, Discrete Appl. Math. 16 (1987) 135–149.

[6] M.L. Fisher, The Lagrangean relaxation method for solving integer programming problems, Management Sci. 27 (1981) 1–18.

[7] G.N. Frederickson and D.B. Johnson, The complexity of selection and ranking in $X+Y$ and matrices with sorted columns, J. Comput. System Sci. 24 (1982) 197–208.

[8] A. Galperin and Z. Waksman, A separable integer programming problem equivalent to its continual version, J. Comput. Appl. Math. 7 (1981) 173–179.

[9] D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees, in: Proceedings 13th Annual A.C.M. Symposium on Theory of Computing (1981) 114–122.

[10] A. Tamir, Further remarks on selection problems with nested constraints, Department of Statistics Report, Tel Aviv University, Tel Aviv (1979).

[11] A. Tamir, Efficient algorithms for a selection problem with nested constraints and its application to a production-sales planning model, SIAM J. Control Optim. 18 (1980) 282–287.

[12] R.E. Tarjan, Data structures and network algorithms, in: CBMS-NSF Regional Conference Series in Applied Mathematics (SIAM, Philadelphia, PA, 1983).