Mathematical Programming 69 (1995) 269–309

About strongly polynomial time algorithms for quadratic optimization over submodular constraints ☆

Dorit S. Hochbaum [a,b,*], Sung-Pil Hong [b]

[a] *School of Business Administration, University of California, Berkeley, CA, USA*
[b] *Department of IEOR, University of California, Berkeley, CA 94720, USA*

Received 17 February 1993; revised manuscript received 18 July 1994

Abstract

We present new strongly polynomial algorithms for special cases of convex separable quadratic minimization over submodular constraints. The main results are: an $O(NM \log(N^2/M))$ algorithm for the problem *Network* defined on a network on M arcs and N nodes; an $O(n \log n)$ algorithm for the *tree* problem on n variables; an $O(n \log n)$ algorithm for the *Nested* problem, and a linear time algorithm for the *Generalized Upper Bound* problem. These algorithms are the best known so far for these problems. The status of the general problem and open questions are presented as well.

Keywords: Quadratic programming; Submodular constraints; Kuhn–Tucker conditions; Lexicographically optimal flow; Parametric maximum flow

1. Introduction

In this paper we investigate strongly polynomial algorithms for convex quadratic optimization problems. The motivation for such study is the fact that a number of classes of Linear Programming are solvable in strongly polynomial time, whereas the corresponding quadratic optimization problems are not known to be solvable in strongly polynomial time. In particular, Tardos proved in [24] that "combinatorial" Linear Programming problems are solvable in strongly polynomial time. (Combinatorial linear

☆ This research has been supported in part by ONR grant N00014-91-J-1241.
* Corresponding author.

SSDI 0025-5610(94)00055-7

programs are those with 0–1 coefficients in the constraint matrix.) Separable convex quadratic optimization problems over "combinatorial" constraints are not known to possess strongly polynomial algorithms.

Solvability in strongly polynomial time for a linear or quadratic programming problem on n variables and m constraints means that there exists an algorithm that solves the problem in a number of steps that is bounded by a polynomial function of n and m only. The general optimization problem over submodular constraints can however be described without an explicit description of the constraints. Such is the case when there is a constraint for each subset of the universal set of n variables. The input then describes the *rank* function defined on all possible subsets of the universal set. In this case, for an algorithm to be strongly polynomial, its running time depends on n alone, and the length of the description of the rank function.

Although combinatorial Linear Programming problems are solvable in strongly polynomial time, this feature is not shared with nonlinear problems. In [17], it was shown that nonquadratic concave separable optimization problems are *not* solvable in strongly polynomial time in a computation model that includes the arithmetic operations, comparisons and the floor operation. This lower bound was illustrated for the *simple resource allocation problem* $\max\{\sum_{j=1}^{n} f_j(x_i) \mid \sum_{j=1}^{n} x_j \leqslant B, \mathbf{x} \geqslant 0, \mathbf{x} \text{ integer}\}$, and for its continuous version. The simple resource allocation problem is the simplest form of nonlinear optimization over submodular constrains. This negative result applies only for nonquadratic objective functions, so the issue of the strong polynomiality of quadratic optimization problems over linear constraints is still open.

While for a general optimization problem there is a clear distinction in complexity between optimizing over integers or over continuous variables, this is not the case for optimization over submodular constraints. It is proved in [17] that there is a "proximity" theorem between an optimal integer and optimal continuous solution to the problem where any optimal continuous solution rounded down bounds from below an integer optimal solution. This allows in particular to solve the integer problem by solving first the continuous problem and then apply what amounts to at most n steps to reach an optimal integer solution. This strategy is adopted throughout this paper in order to derive continuous *and* integer solutions to the quadratic optimization problem over submodular constraints.

Known cases where convex quadratic optimization in integers (or continuous variables) over linear constraints can be solved in strongly polynomial time include: a nonseparable quadratic transportation problem [19]; an unconstrained nonseparable quadratic optimization in the context of electrical distribution system [1]; a nonseparable problem in the context of toxic waste disposal [15]; a quadratic continuous Knapsack problem [4]; a problem where the constraints consist of two equations and lower and upper bounds [2]; a transportation problem with fixed number of sources (or sinks) [6]; an improvement in complexity to the transportation problem with fixed number of sources and extending the strong polynomiality to a quadratic problem over a fixed number of equations [21]; a quadratic series-parallel network with a single source and sink [23].

Our aim in this paper is to establish the most efficient strongly polynomial algorithms known for several quadratic problems over submodular constraints. The general quadratic problem over submodular constraints is defined with respect to some *submodular* rank function $r: \Lambda \to R$ defined on a distributive lattice Λ of $E = \{1, \ldots, n\}$ (a set of subsets of E which contains $\emptyset$, E and is closed on the set intersection and union), i.e. $r(\emptyset) = 0$ and for all $A, B \in \Lambda$,

$$r(A) + r(B) \geqslant r(A \cup B) + r(A \cap B).$$

(For a detailed description of submodular functions see e.g. [22].) The *submodular polyhedron* defined by the submodular function r is the set $\{\mathbf{x} | \sum_{j \in A} x_j \leqslant r(A),\ A \in \Lambda\}$.

We call the system of inequalities $\{\sum_{j \in A} x_j \leqslant r(A) | A \in \Lambda\}$, *submodular constraints*. The problem of quadratic integer optimization over submodular constraints is then,

$$\begin{aligned} \min \quad & \sum_{j \in E} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\ & \sum_{j \in A} x_j \leqslant r(A), \quad A \in \Lambda \\ & x_j \geqslant 0 \text{ and integer}, \quad j \in E. \end{aligned}$$

For $\mathbf{b}$ a nonnegative vector, the objective function is convex. This is a special case of the convex nonlinear problem over submodular constraints, the *general resource allocation problem* or *(GAP)*:

$$\begin{aligned} (GAP) \qquad \max \quad & \sum_{j \in E} f_j(x_j) \\ & \sum_{j \in A} x_j \leqslant r(A), \quad A \in \Lambda \\ & x_j \geqslant 0 \text{ and integer}, \quad j \in E. \end{aligned}$$

The problem *(GAP)* was proved polynomial by Groenevelt [16], using the ellipsoid algorithm, and by Hochbaum [17] using a proximity and scaling based algorithm. Since the number of constraints in the problem could be exponential in $|E|$, the running time is expressed in terms of the number of calls to an oracle that determines whether a solution is a member of the submodular polyhedron, or equivalently, feasible for the submodular constraints. Let F denote the number of steps that an oracle requires to determine whether incrementing a given feasible solution vector in one of its components by one unit results in a feasible solution vector. The running time given in [17] is $O(n(\log n + F) \log_2(r(E)/n)))$, and for the continuous case an ε-accurate solution (within ε in the solution space) is produced in $O(n(\log n + F) \log_2(r(E)/\varepsilon n))$ steps. (There is no statement of running time in [16].) Note that this running time is polynomial, but not strongly polynomial as it depends on the value of the right-hand side, $r(E)$. These algorithms apply particularly to the problem of quadratic optimization over submodular constraints.

The problem (*GAP*) has been studied extensively in the literature. A book by Ibaraki and Katoh [20] presents an excellent state-of-the-art survey on this problem and its special cases. Here we focus on the instances of the problem where the objective function is quadratic. We present here strongly polynomial algorithms for all cases of (*GAP*) studied in the literature. These problems, in addition to the simple resource allocation problem, (*SRA*), are the *generalized upper bound* resource allocation problem, (*GUB*), the *nested* resource allocation problem, (*Nested*), the *tree* resource allocation problem, (*Tree*), and the *network* resource allocation problem, (*Network*). The definitions and formulations of these problems are given in Section 2.

Prior work on strongly polynomial algorithms for the problems discussed here includes two algorithms. In [9], Fujishige devised an algorithm for the *lexicographically optimum flow problem* from which it is possible to derive an $O(N^2M \log(N^2/M))$ time algorithm for (*Network*) when the underlying network has M arcs and N nodes, and hence for all other problems described here. This algorithm, when applied to (*Tree*), runs in $O(n^2)$ time. Another algorithm by Tamir [23], solves the minimum convex separable quadratic cost flow problem on series-parallel network for which (*Tree*) is a special case. Applied to the problem (*Tree*), this algorithm has complexity $O(n^2)$.

The main results here are an $O(NM \log(N^2/M))$ algorithm for (*Network*), an $O(n \log n)$ algorithm for (*Tree*) on n variables, an $O(n)$ algorithm for (*Nested*) when given a sorted array of the coefficients, a_j, and a linear time algorithm for (*GUB*). These results constitute therefore a significant improvement on the complexity of currently available algorithms. Such efficient algorithms also lend additional support to the conjecture that the problem of quadratic cost network flow is solvable in strongly polynomial time.

The paper is organized as follows. Section 2 defines the classes of problems addressed and gives their formulations. In Section 3 we give the algorithm for the quadratic simple resource allocation problem that is used as building blocks for the nested and tree algorithms. Section 3 also describes the linear time algorithm used for the generalized upper bounds problem. Section 4 contains the algorithm for (*Nested*), and Section 5 contains the algorithm for (*Tree*). Section 6 describes the algorithm used for the (*Network*) case, and our implementation of a parametric flow algorithm.

2. Formulations and preliminaries

2.1. Formulations

Important special cases of optimization over submodular constraints that have been studied in the literature are formulated here with a quadratic objective function. The formulations given here include a constraint for the rank of the entire set as an inequality constraint. However, all the algorithms given in this paper can be easily modified to

solve the corresponding problem given with this constraint as an equality. The problems are also slightly generalized by allowing upper bound constraints on the variables.

We assume throughout that the objective functions are strictly convex, that is, the vector **b** is positive. This assumption is made for the sake of convenience of the presentation. All algorithms described apply also when some of the functions are linear with an obvious modification. For the network problem, where the modification is less obvious, there is a discussion on the method of modifying the algorithm. We choose not to treat the non-strictly-convex cases explicitly, in order not to obscure the main algorithmic issues involved.

(1) The *simple* resource allocation problem:

$$
\begin{aligned}
(SRA) \qquad \min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=1}^{n} x_j \leqslant B \\
& 0 \leqslant x_j \leqslant u_j \text{ integers}, \quad j = 1, \ldots, n.
\end{aligned}
$$

The problem (*SRA*) may be viewed as a minimum cost flow problem with a source that has supply of B units. Each variable represents the amount of flow along each arc going from a node to a sink t. There are no costs or capacities associated with the arcs going into the nodes other than the sink, but there are capacity upper bounds u_j associated with the arc going from node j to the sink and also quadratic cost functions $a_j x_j + \frac{1}{2} b_j x_j^2$. Since the "supply" in the formulation above is *up to* B, this can be incorporated by adding an arc with zero cost (and infinite capacity) from source to t. Such arc is omitted from the network described in Fig. 1(a). Note that (*SRA*) could also be considered as a quadratic transportation problem with a single supplier and n customers. This observation underlied the technique used in [6] for solving quadratic transportation problems.

(2) The *generalized upper bound* resource allocation problem:

$$
\begin{aligned}
(GUB) \qquad \min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=1}^{n} x_j \leqslant B \\
& \sum_{j \in S_i} x_j \leqslant p_i, \quad i = 1, \ldots, m \\
& 0 \leqslant x_j \leqslant u_j \text{ integers}, \quad j = 1, \ldots, n.
\end{aligned}
$$

where $\{S_1, S_2, \ldots, S_m\}$ is a partition of $E = \{1, \ldots, n\}$, i.e. disjoint sets the union of which is E. A depiction of this problem as a minimum cost flow problem is given in Fig. 1(b).

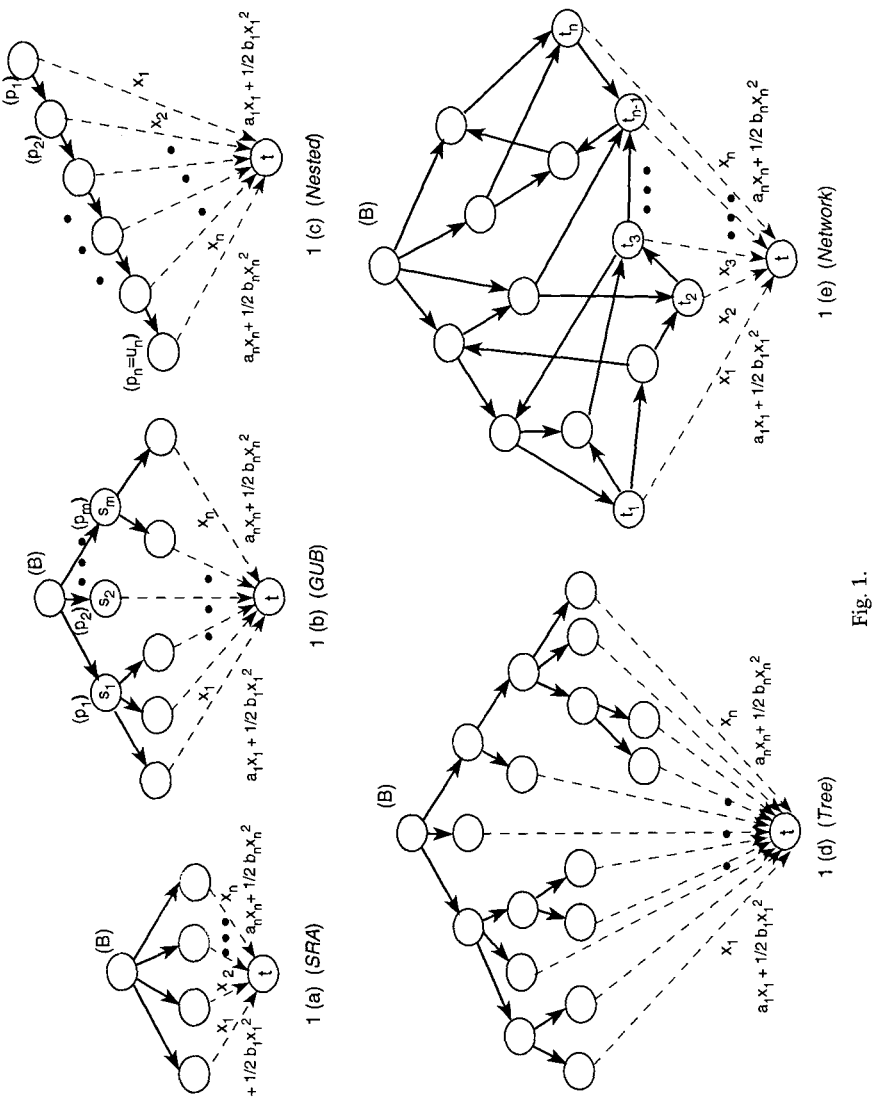

Fig. 1.

(3) The *nested* resource allocation problem:

$$
\begin{aligned}
(\textit{Nested}) \qquad \min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=1}^{n} x_j \leqslant B \\
& \sum_{j \in S_i} x_j \leqslant p_i, \quad i = 1, \ldots, m \\
& 0 \leqslant x_j \leqslant u_j \text{ integers}, \quad j = 1, \ldots, n.
\end{aligned}
$$

where $S_m \subset S_{m-1} \subset \cdots \subset S_1 \subset E$. Notice that $p_m \leqslant p_{m-1} \leqslant \cdots \leqslant p_1$, otherwise, if $p_i > p_{i+1}$, then the ith constraint is redundant and can be omitted.

It is more convenient to analyze (*Nested*) with a constraint corresponding to each variable, that is, it is always possible to reduce (*Nested*) to the following special case with set $S_i = \{i, i+1, \ldots, n\}$. Here $p_1 = B$.

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=i}^{n} x_j \leqslant p_i, \quad i = 1, \ldots, n \\
& 0 \leqslant x_j \leqslant u_j \text{ integers}, \quad j = 1, \ldots, n.
\end{aligned}
$$

If a set $\{x_i, \ldots, x_n\}$ does not appear among the sets S_j, then set its right-hand side p_i to be equal to that of the smallest set among the S_j's containing $\{x_i, \ldots, x_n\}$.

The problem (*Nested*) is described in Fig. 1(c).

(4) The *tree* resource allocation problem:

$$
\begin{aligned}
(\textit{Tree}) \qquad \min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=1}^{n} x_j \leqslant B \\
& \sum_{j \in S_i} x_j \leqslant p_i, \quad i = 1, \ldots, m \\
& 0 \leqslant x_j \leqslant u_j \text{ integers}, \quad j = 1, \ldots, n.
\end{aligned}
$$

The sets S_i are derived by some hierarchical decomposition of E into disjoint subsets and the repeated decomposition of each of the subsets. Each set thus generated is among the sets S_i, $i = 1, \ldots, m$. Describing each set as a node and the decomposition as edges from the parent set to its subsets, one gets a tree on m nodes which is a branching, i.e. the indegree of each node except the root corresponding to the set E is one.

It is convenient to extend the tree of sets by adding all singleton sets as leaves. Note

that the problem could be viewed as flow problem from the root to the leaves where the objective function minimizes the quadratic cost of the flow to the leaves only. All other flows have cost of zero, and only the capacitated nodes and the flow balance constraints determine the feasibility. The network describing the flow problem corresponding to the tree allocation problem is given in Fig. 1(d).

(5) The *network* resource allocation problem is defined with respect to any network (or graph), with a single source and a set of sinks.

Given a directed graph (network) $G=(V, A)$ with node set V and arc set A. Let $s \in V$ be the source and $T \subseteq V$ be the set of sinks. The supply of the source is $B>0$, and the capacity of arc (i, j) is c_{ij}. Denote the flow vector by $\mathbf{f}=\{f_{ij} \mid (i, j) \in A\}$

$$
\begin{aligned}
(\textit{Network}) \qquad \min \quad & \sum_{t_k \in T} a_k x_k + \tfrac{1}{2} b_k x_k^2 \\
& \sum_{(i,j)\in A} f_{ij} - \sum_{(j,i)\in A} f_{ji} = 0, \quad i \in V - T - \{s\} \\
& \sum_{(s,j)\in A} f_{sj} \leqslant B \\
& \sum_{(j,t_k)\in A} f_{jt_k} - \sum_{(t_k,j)\in A} f_{t_kj} = x_k, \quad t_k \in T \\
& 0 \leqslant f_{ij} \leqslant c_{ij}, \quad (i, j) \in A \\
& 0 \leqslant x_k \leqslant u_k, \quad t_k \in T.
\end{aligned}
$$

Given a feasible flow $\mathbf{f}$ in G, each variable x_k represents the net value of the flow arriving at the sink t_k. We call $\mathbf{x}$ the *out-flow* vector of the flow $\mathbf{f}$. The (quadratic) network resource allocation problem (*Network*) is not the same problem as the minimum quadratic cost flow problem. In the latter problem there is an underlying graph and a quadratic cost associated with the flow along each arc. In this problem the quadratic costs are associated only with the flow x_k arriving at each sink t_k. An alternative representation is to augment G with a dummy sink t, and connect each sink t_k to t with a directed arc (t_k, t) of capacity u_k. The costs are then only associated with the arcs (t_k, t). All other arcs have 0 cost associated with them. This graph is described in Fig. 1(e).

The relations between these problems are depicted in Fig. 2 where $A \rightarrow B$ if problem A is a special case of problem B.

2.2. Deriving integer from continuous solutions

Vectors in this paper are denoted by boldface letters. The vector $\mathbf{e}$ denotes the n-vector $(1, \ldots, 1)$.

A theorem in [17] states a proximity between an optimal (integer) solution to (*GAP*) and a scaled solution. A corollary of this theorem is a proximity result on the distance between an optimal integer and optimal continuous solutions to (*GAP*). Such result is

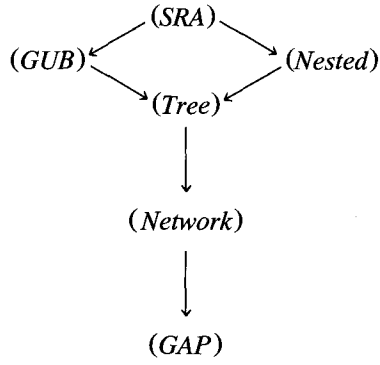

Fig. 2.

not useful in finding optimal integer solutions to the problem unless the continuous problem is particularly easy to solve. The statement of this corollary [17, Corollary 4.4] is:

Corollary 2.1. *For an integer optimal solution to (GAP), $\mathbf{z}^*$, there is a continuous optimal solution to (GAP), $\mathbf{x}^*$, such that $\mathbf{z}^* - \mathbf{e} < \mathbf{x}^* < \mathbf{z}^* + n\mathbf{e}$, and vice versa; i.e. for a continuous optimal solution to (GAP), $\mathbf{x}^*$, there is an integer optimal solution to (GAP), $\mathbf{z}^*$, such that $\mathbf{z}^* - \mathbf{e} < \mathbf{x}^* < \mathbf{z}^* + n\mathbf{e}$.*

In particular, $\|\mathbf{z}^* - \mathbf{x}^*\|_\infty < n$. This is a tighter proximity theorem than the one existing in the literature for constrained linear [5], quadratic [14] and nonlinear [16] optimization problems, all of which have $\|\mathbf{z}^* - \mathbf{x}^*\|_\infty < n\Delta$, where Δ is the largest subdeterminant of the constraint matrix. This result could be viewed as effectively considering the largest subdeterminant of a set of submodular constraints to be 1, although such subdeterminant is in general exponentially large.

The proximity theorem is used to produce more efficiently integer solutions to the quadratic cases of (*GAP*), where the continuous solution is relatively easy to derive from Kuhn–Tucker conditions (all of which are linear for quadratic objective function): First a continuous solution is obtained, $\mathbf{x}^*$. The vector $\bar{\mathbf{x}} = [\mathbf{x}^* + \mathbf{e}]$ is then an upper bound on an integer optimal solution and the sum of its components is at most $r(E) + n$. Hence it suffices to remove the, up to n, units of $\bar{\mathbf{x}}$ that contribute least to the objective function. This is done by considering the incremental contribution of each last unit of each component and removing the one that reduces the objective function by the least amount. This is continued until the constraints including $\sum_{j=1}^{n} x_j \leqslant B$ are satisfied. The validity of such a greedy approach is documented in [2o] and in [17].

Although it appears that a direct implementation of the procedure above requires $O(n^2)$ time even for (*SRA*), this is not the case. The problem of obtaining an optimal integer solution from $\bar{\mathbf{x}}$ is also an allocation problem in integers, but with right-hand

sides that are $O(n)$. Such allocation problems, if they have same constraints as the problems in Subsection 2.1, with any convex separable objective function, are solvable with the following running times:

(*SRA*) in $O(n)$, [8],

(*GUB*) in $O(n)$, [17],

(*Nested*) in $O(n \log n)$ [17],

(*Tree*) in $O(n \log n)$ [17],

(*Network*) and any submodular constraints, in $O(nF)$ where F is the number of steps required to check whether an increment of one unit (of flow, in (*Network*)) is feasible.

These running times are added to the complexity of the continuous problem in order to determine the complexity of the integer problem. Yet, in all cases these running times are dominated by those required to solve the continuous problem. Thus in the subsequent part of this paper, we consider only the continuous versions of the problems defined in the previous subsection.

3. A linear time algorithm for (*SRA*) and (*GUB*)

Brucker [4], was the first to devise a linear time algorithm for the continuous convex quadratic Knapsack problem. This problem is more general than (*SRA*) in that its constraint may have nonnegative coefficients to the variables where in (*SRA*) all these coefficients are 1. The algorithm presented here is also directly applicable to the Knapsack version of the problem with a minor adjustment. The presentation here follows the algorithm given in [6] with some appropriate modifications.

At the optimum of (*SRA*) the derivative with respect to each variable has to be nonpositive. (Otherwise, a variable with positive derivative value at the optimum can be decreased by $\varepsilon > 0$ while only improving (reducing) the objective function and without violating any constraint.) In other words, $x_j \leqslant \max\{0, -a_j/b_j\}$. Hence we can update $u_j \leftarrow \min\{u_j, \max\{0, -a_j/b_j\}\}$ for each j and then $B \leftarrow \min\{B, \sum_{j=1}^{n} u_j\}$ without affecting the optimality. Also, notice that with this preprocessing, taking $O(n)$ time, at any optimal solution, $\sum_{j=1}^{n} x_i \leqslant B$ is binding. Otherwise there should be a variable with value less than the (updated) upper bound and hence with a negative derivative. Then we can increase the value of the variable by a small amount to reduce the objective value while maintaining the feasibility, contradicting to the optimality assumption. Thus by a linear time preprocessing, (*SRA*) is reducible to the same problem with equality constraint:

$$
\begin{aligned}
(SRA) \qquad \min \quad & \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=1}^{n} x_j = B \\
& 0 \leqslant x_j \leqslant u_j, \quad j = 1, \ldots, n,
\end{aligned}
$$

where B is positive, $B \leqslant \sum_{j=1}^{n} u_j$ and each b_j is positive.

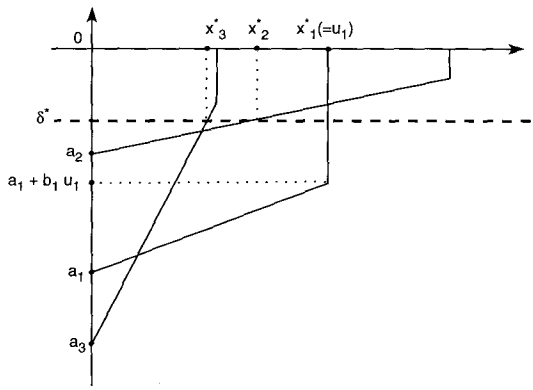

Fig. 3.

The convexity of the objective function guarantees that a solution satisfying the Kuhn–Tucker conditions is also optimal. In particular, we seek a nonnegative solution $\mathbf{x}^*$ and a value δ^* such that

$$\sum_{j=1}^{n} x_j^* = B \quad \text{and} \quad u_j > x_j^* > 0 \Rightarrow a_j + b_j x_j^* = \delta^*.$$

The situation is illustrated by Fig. 3.

The value set for δ determines associated values for x_j. For any value δ, the associated solution $\mathbf{x}(\delta)$ is

$$x_j(\delta) = \begin{cases} 0 & \text{if } \delta \leqslant a_j, \\ (\delta - a_j)/b_j & \text{if } a_j < \delta \leqslant a_j + b_j u_j, \\ u_j & \text{if } a_j + b_j u_j < \delta. \end{cases} \tag{3.1}$$

Let $\hat{B}(\delta) = \sum_{j=1}^{n} x_j(\delta)$. Then finding the optimal solution to (*SRA*) is equivalent to finding a value δ^* such that $\hat{B}(\delta^*) = B$. (Since $0 < B \leqslant \sum_{j=1}^{n} u_j$, it follows that there is a finite optimal δ^* for every instance of (*SRA*).)

Notice that $\hat{B}(\delta)$ is a monotone increasing, piecewise linear function of δ, having breakpoints at the values a_i, and $a_j + b_j u_j$ for $j = 1, \ldots, n$. So if $\hat{B}(\delta) < B$, then we could conclude that δ^* is greater than δ and similarly, if $\hat{B}(\delta) > B$, then δ^* is less than δ. Thus the monotonicity of $\hat{B}(\delta)$ allows for a binary search for the optimal value, δ^* satisfying $\hat{B}(\delta) = B$.

The algorithm we propose for finding δ^*, chooses "guesses" (from among the breakpoint values, a_j and $a_j + b_j u_j$) until it finds two consecutive breakpoints which contain δ^* in the interval between them. In this range, $\hat{B}(\delta)$ is a linear function. The problem is then solved by finding the particular value of δ for which $\hat{B}(\delta) = B$ (i.e., by solving the linear equation in one variable).

From (3.1), we have,

$$\hat{B}(\delta) = \delta \sum_{j \in U} \frac{1}{b_j} - \sum_{j \in U} \frac{a_j}{b_j} + \sum_{j \in V} u_j, \tag{3.2}$$

where $U = \{j \mid a_j < \delta \leqslant a_j + b_j u_j\}$ and $V = \{j \mid a_j + b_j u_j < \delta\}$.

So at each iteration, we need to determine the index sets and the corresponding sums. To result in a better complexity, the algorithm avoids computing the index sets and the sums at every iteration from scratch. For this purpose, it maintains the parameters P, Q and R which retain partial sums from the previous iteration.

Procedure SRA

Step 0: {initialization} $S \leftarrow \{a_1, \ldots, a_n;\ a_1 + b_1 u_1, \ldots, a_n + b_n u_n\}$.
$I,\ J \leftarrow \{1, \ldots, n\}$, $P \leftarrow \sum_{j=1}^{n} a_j / b_j$, $Q \leftarrow \sum_{j=1}^{n} 1/b_j$.

Step 1: {selecting median of breakpoints}
Set $\hat{\delta}$ to be the median value from the set S.
{computing coefficients of $\hat{B}(\hat{\delta})$}
$L(\hat{\delta}) \leftarrow \{j \in I \mid \hat{\delta} \leqslant a_j\}$, $R(\hat{\delta}) \leftarrow \{j \in J \mid a_j + b_j u_j < \hat{\delta}\}$,
$M(\hat{\delta}) \leftarrow L(\hat{\delta}) \cup R(\hat{\delta})$
$\hat{P} \leftarrow P - \sum_{j \in M(\hat{\delta})} a_j / b_j$, $\hat{Q} \leftarrow Q - \sum_{j \in M(\hat{\delta})} 1/b_j$, $\hat{R} \leftarrow \sum_{j \in R(\hat{\delta})} u_j$.

Step 2: {computing $\hat{B}(\hat{\delta})$} $\hat{B} \leftarrow \hat{\delta}\hat{Q} - \hat{P} + \hat{R}$.
If $\hat{B} \leftarrow B$ then STOP, $\delta^* \leftarrow \hat{\delta}$.
If $\hat{B} > B$ then $\delta^* < \hat{\delta}$.
If $\hat{B} < B$ then $\delta^* > \hat{\delta}$.

Step 3: {update index sets, breakpoints and partial sums}
If $\delta^* < \hat{\delta}$ then
$I \leftarrow I - L(\hat{\delta})$, $J \leftarrow R(\hat{\delta})$, $S \leftarrow \{a_j \mid j \in I\} \cup \{a_j + b_j u_j \mid j \in J\}$.
If $\hat{\delta} = a_m$ for some m, then $P \leftarrow \hat{P} - a_m / b_m$, $Q \leftarrow \hat{Q} - 1/b_m$.
Else $\{\delta^* > \hat{\delta}\}$,
$I \leftarrow L(\hat{\delta})$, $J \leftarrow J - R(\hat{\delta})$, $S \leftarrow \{a_j \mid j \in I\} \cup \{a_j + b_j u_j \mid j \in J\}$.

Step 4: {repeating until final interval is found}
If $|S| \geqslant 2$, go to Step 1.
Else, $\delta^* \leftarrow (B + \hat{P} - \hat{R})/\hat{Q}$.

The algorithm outputs a value δ^*. Then the optimal solution is $\mathbf{x}(\delta^*)$ which can be determined in linear time using (3.1).

Theorem 3.1. *Procedure* **SRA** *finds* δ^* *and* $\mathbf{x}^*$ *in* $O(n)$ *time.*

Proof. To prove the validity of Procedure **SRA**, we need to show the correctness of $\hat{B}$ in Step 2, which is the value of $\hat{B}(\delta)$ for $\delta = \hat{\delta}$.

Consider $\hat{Q}$ in Step 1, which is the slope of the piecewise linear curve, $\hat{B}(\delta)$, at $\delta = \hat{\delta}$. To compute $\hat{Q}$, we first calculate Q which represent the maximum possible value

of the slope of $\hat{B}(\delta)$ when δ takes value among the breakpoints of S. Thus, initially Q is $\sum_{j=1}^{n} 1/b_j$. From (3.2) and the definitions of $L(\hat{\delta})$ and $R(\hat{\delta})$, it follows that $M(\hat{\delta})$ is the set of indices j such that $1/b_j$ needs to be subtracted from Q to obtain the correct slope of $\hat{B}(\delta)$ for $\delta = \hat{\delta}$. Therefore, $\hat{Q}$ should be

$$\sum_{j \in U} \frac{1}{b_j} = \sum_{j=1}^{n} \frac{1}{b_j} - \sum_{j \in M(\delta)} \frac{1}{b_j} = Q - \sum_{j \in M(\hat{\delta})}^{n} \frac{1}{b_j}. \tag{3.3}$$

Hence the coefficient $\hat{Q}$ calculated in Step 1 is correct for the first iteration.

If $\delta^* < \hat{\delta}$, then the next guess is the median of the lower half of the current breakpoints, that is, those breakpoints less than $\hat{\delta}$. So in Step 3, the upper half of the current breakpoints (including the current guess $\hat{\delta}$) is deleted from the set S and in Step 3 S is updated accordingly. In this case Q, the maximum possible slope of $\hat{B}(\delta)$ over the updated S, is $\hat{Q} - a_j/b_m$ if $\hat{\delta} = a_m$ for some m, or $\hat{Q}$, otherwise. Furthermore, from the updated set I and J in Step 3 it follows again that $M(\hat{\delta})$ is the set of indices j such that $1/b_j$ needs to be subtracted from (the updated) Q to obtain the correct slope of $\hat{B}(\delta)$ for $\delta = \hat{\delta}$ in the next iteration. Thus the correctness of $\hat{Q}$ obtained in Step 1 follows by induction on the numbers of iteration.

On the other hand, if $\delta^* > \hat{\delta}$, then the next guess is the median of the upper half of the current breakpoints. So, in this case, we use the same Q in the next iteration. Similar inductive arguments for $\hat{P}$ and $\hat{R}$ show the correctness of the computation of $\hat{B}(\hat{\delta})$ in Step 2.

When S contains only one element, say a_j (or, $a_j + b_j u_j$), then we can conclude that δ^* is between $\hat{\delta}$ and a_j (or, $a_j + b_j u_j$, respectively). Furthermore, since $\hat{B}$ is a linear function of δ in this range, (i.e. $\hat{B} = \hat{Q}\delta - \hat{P} + \hat{R}$), δ^* and $\mathbf{x}^*$ are determined as in Step 4.

The $O(n)$ complexity of the algorithm follows from the fact that each of Step 1, 2 and 3 can be performed in a number of arithmetic operations that is linear in the cardinality of the set S, including the selection of the median value [3]. Since the number of elements in the set is initially $2n$ and is cut in half after each pass, the total work is linear in $(2n + n + n/2 + n/4 + \cdots) = 4n$, so the complexity of the algorithm is $O(n)$. □

The problem (GUB) is easier to handle once we observe that it is polynomially equivalent to a number of simple resource allocation problems. Consider the set S_i, the constraint $\sum_{j \in S_i} x_j \leqslant p_i$, and the following simple resource allocation problem restricted to S_i:

$$(SRA_i) \qquad \min \quad \sum_{j \in S_i} a_j x_j + \tfrac{1}{2} b_j x_j^2$$

$$\sum_{j \in S_i} x_j \leqslant p_i$$

$$0 \leqslant x_j, \quad j \in S_i.$$

Lemma 3.2 (Hochbaum [17]). *Let the solution to* (SRA_i) *be* $\{x_j^{(i)}\}_{j \in S_i}$. *Then there exists an optimal solution to* (GUB), $\mathbf{x}^*$, *satisfying* $x_j^* \leqslant x_j^{(i)}$ *for all* $j \in S_i$.

Remark. In [17], the lemma is proved for the discrete version of the problem. It is easy to see the proof is modified for the continuous version of the problem. This lemma is generalized and proved for the (continuous) (*Tree*) problem in Corollary 5.2.

Lemma 3.2 implies that once an optimal solution, $\{x_j^{(i)}\}_{j \in S_i}$ has been obtained for (SRA_i) for every $i = 1, \ldots, m$, an optimal solution of (GUB) can be found by solving the following problem, (UB), which is also an (SRA).

$$(UB) \qquad \min \quad \sum_{j=1}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2$$

$$\sum_{j=1}^{n} x_j = B$$

$$0 \leqslant x_j \leqslant \min\{u_j, x_j^{(i)}\}, \quad j = 1, \ldots, n.$$

It is therefore sufficient to solve each of the (SRA_i) problems, in order to derive the upper bounds. Then to solve the problem (UB). The running time of such procedure is $O(n_1) + O(n_2) + \cdots + O(n_m) = O(n)$, followed by the linear time required to solve the resulting (UB) (which is an (SRA)).

4. An $O(n \log n)$ algorithm for (*Nested*)

The algorithm proposed here solves the problems $(Nested_n)$, $(Nested_{n-1}), \ldots,$ $(Nested_1)$, where $(Nested_i)$ is the problem,

$$(Nested_i), \qquad \min \quad \sum_{j=i}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2$$

$$\sum_{j=k}^{n} x_j \leqslant p_k, \quad k = i, \ldots, n$$

$$0 \leqslant x_j \leqslant u_j, \quad j = i, \ldots, n.$$

Let an optimal solution to $(Nested_i)$ be $\mathbf{x}^{(i)}$. Several properties of $(Nested_i)$ are essential in order to establish the correctness of the algorithm. The next lemma states that for the problem $(Nested_i)$, the constraint $\sum_{j=i}^{n} x_j^{(i)} \leqslant p_i$ may be assumed to be satisfied with equality. The proof is given for the analogous lemma, Lemma 5.3 for the tree resource allocation problem, (*Tree*), which generalizes (*Nested*).

Lemma 4.1. *In* $(Nested_i)$, *by updating* $u_j \leftarrow \min\{u_j, \max\{-a_j/b_j, 0\}\}$ *for* $j = n, n-1, \ldots, 1$ *and* $p_j \leftarrow \min\{p_j, p_{j+1} + u_j\}$ *for* $j = n-1, n-2, \ldots, 1$, *we may assume that the constraint* $\sum_{j=i}^{n} x_j^{(i)} \leqslant p_i$ *is satisfied with equality.*

The following lemma contains the key idea of the algorithm. The proof is postponed to Section 5 where it appears as Corollary 5.2.

Lemma 4.2. $x_j^{(i)} \leqslant x_j^{(i+1)}$, *for* $j = i+1, \ldots, n$.

This lemma implies that the value of the optimal solution for $(Nested_{i+1})$ is an upper bound on the value of the variables, $x_{i+1}, x_{i+2}, \ldots, x_n$ in $(Nested_i)$. The upper bounds at each $(Nested_i)$ problem solved, u_j can then be updated to $u_j^{(i)} = \min\{u_j, x_j^{(i+1)}\}$, for $j = i+1, \ldots, n$. Since $\{x_j^{(i+1)}\}$ satisfy constraints $i+1, \ldots, n$, these constraints no longer need to be explicitly incorporated. Hence $(Nested_i)$ is equivalent to the problem

$$
\begin{aligned}
(Nested_i) \qquad \min \quad & \sum_{j=i}^{n} a_j x_j + \tfrac{1}{2} b_j x_j^2 \\
& \sum_{j=i}^{n} x_j = p_i \\
& 0 \leqslant x_i \leqslant u_i \\
& 0 \leqslant x_j \leqslant \min\left\{u_j, x_j^{(i+1)}\right\}, \quad j = i+1, \ldots, n.
\end{aligned}
$$

This latter formulation of the problem is an (*SRA*). The algorithm solves recursively the problems $(Nested_i)$ for $i = n, \ldots, 1$ where at each call the optimal solution derived from the previous call is used as upper bounds to the variables in the current call. The optimal solution of $(Nested_{i+1})$ is then used to derive an optimal solution to $(Nested_i)$ in constant amortized running time.

The problem $(Nested_i)$ is an (*SRA*). This suggests immediately an algorithm that requires linear time with each call using Procedure **SRA**. Such algorithm would result in complexity of $O(n^2)$. In order to get a more efficient approach, we maintain all information obtained in previous iterations on the status of the breakpoints previously considered. These breakpoints are also maintained in a sorted array. The need to maintain a sorted array adds an additive factor of $O(n \log n)$ to the running time at a preprocessing step. The algorithm runs in linear time when the sorted sequence of coefficients $\{a_1, \ldots, a_n\}$ is available with the input.

The algorithm produces a Lagrange multipliers δ for each (*SRA*), $(Nested_i)$ for $i = n, \ldots, 1$. Let the Lagrange multiplier for $(Nested_i)$ be δ_i. Unlike Procedure **SRA** which finds δ by binary search, testing $\hat{B}(\delta)$ on the median of the current (unsorted) breakpoints, the algorithm finds δ by "linear search", testing $\hat{B}(\delta)$ on consecutive elements of the current breakpoints given in sorted array. That is, starting with an initial guess, it continues to test the immediate successor (or predecessor) of the current guess until it finds the final interval.

The initial guess for δ_i is δ_{i+1}. All variables x_j for $j = i+1, \ldots, n$ are fixed for any $\delta \geqslant \delta_{i+1}$ in testing since, by Lemma 4.2, the optimal solution of $(Nested_i)$ is bounded by the optimal solution of $(Nested_{i+1})$. So if $\delta_i > \delta_{i+1}$, then there are only $O(1)$ arithmetic calculations required to find δ_i as there are at most two breakpoints to be tested; namely, a_i and $a_i + b_i u_i$. δ_i is then added to the top of the sorted sequence of breakpoints for the testing to solve $(Nested_{i-1})$. The crucial property is that when

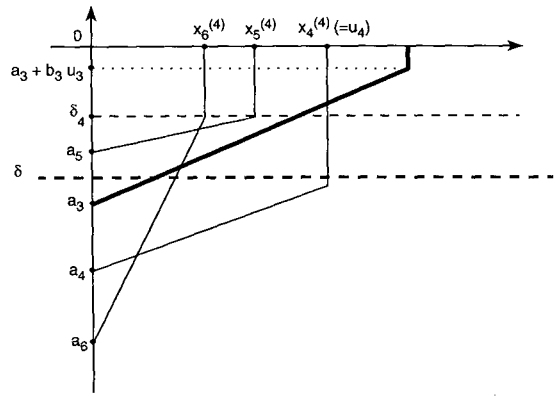

Fig. 4. Solving $(Nested_3)$ given the solution of $(Nested_4)$ where $n = 6$.

$\delta_i < \delta_{i+1}$ then all breakpoints tested with value v such that $\delta_i < v$ need no longer be considered when we solve $(Nested_{i-1})$. For, by Lemma 4.2 again, all variables $x_i, \ldots, x_n$ are fixed for $\delta \geqslant \delta_i$ when we test $B(\delta)$ to solve $(Nested_{i-1})$. Thus those breakpoints are deleted from the sorted sequence. The breakpoint arrangement at a typical iteration is depicted in Fig. 4. The thick solid line represents the piecewise linear curve corresponding to the additional variable x_3 of $(Nested_3)$. (Compare this figure to Fig. 3.)

In the algorithm there is some information stored at each δ breakpoint. $U(\delta)$ is the sum of all variables whose upper bounds are attained at a breakpoint lower than δ. $P(\delta)$ and $Q(\delta)$ are sums of a_j/b_j and $1/b_j$ respectively over variables j that get fixed at the breakpoint δ. For $(Nested_i)$ these are all the variables with index in the set $\{i, \ldots, n\}$ that are not summed up in $U(\delta)$. For each variable in this sum its upper bound at the termination of this iteration is $\delta/b_j - a_j/b_j$.

The description of Algorithm **Nested** is followed by the description of Procedure **Nested(i)** called for in Step 2 of the algorithm.

Algorithm Nested

Step 0: {Preprocess}
For $j = n, n-1, \ldots, 1$, $u_j \leftarrow \min\{u_j, \max\{-a_j/b_j, 0\}\}$.
For $j = n-1, n-2, \ldots, 1$, $p_j \leftarrow \min\{p_j, p_{j+1} + u_j\}$.
Sort $\{a_1, \ldots, a_n; a_1 + b_1 u_1, \ldots, a_n + b_n u_n\}$ in decreasing order.
Step 1: Solve $(Nested_n)$ by solving for δ, $(\delta - a_n)/b_n = p_n$; $\delta_n \leftarrow \delta$.
$P(\delta_n) \leftarrow a_n/b_n$, $Q(\delta_n) \leftarrow 1/b_n$, $U(\delta_n) \leftarrow 0$; $S \leftarrow \{a_n\}$, $T \leftarrow \{\delta_n\}$.
Step 2: For $i = n-1, \ldots, 1$, call Procedure **Nested(i)**.
Step 3: {calculating optimal solutions using the Lagrange multipliers}

Let the set of the Lagrange multipliers, T, be $\{\delta_{i_1}, \delta_{i_2}, \ldots, \delta_{i_q}\}$; $i_{q+1} \leftarrow n+1$.
For $p = q, q-1, \ldots, 1$, do
$x_j = \max\{\delta_{i_p}/b_j - a_j/b_j, 0\}$ for $j = i_p, \ldots, i_{p+1} - 1$.

Procedure Nested(i)

Input: Two sorted sequences S and T.
For each δ_j, the values of $P(\delta_j)$, $Q(\delta_j)$ and $U(\delta_j)$.

Step 0: {trivial case} If $a_i > 0$; stop, $\delta_i = \delta_{i+1}$.

Step 1: {Check if δ_i is larger or smaller than δ_{i+1}}
If $p_{i+1} + \min\{(\delta_{i+1} - a_i)/b_i, u_i\} < p_i$, then $\delta_i > \delta_{i+1}$; go to Step 2.
If $p_{i+1} + \min\{(\delta_{i+1} - a_i)/b_i, u_i\} = p_i$, then $\delta_i = \delta_{i+1}$; substitute the breakpoint label δ_i by δ_{i+1};
$P(\delta_i) \leftarrow P(\delta_{i+1}) + a_i/b_i$, $Q(\delta_i) \leftarrow Q(\delta_{i+1}) + 1/b_i$, $U(\delta_i) \leftarrow U(\delta_{i+1})$; stop.
If $p_{i+1} + \min\{(\delta_{i+1} - a_i)/b_i, u_i\} > p_i$, then $\delta_i < \delta_{i+1}$; remove δ_{i+1} from the top of T;
$P \leftarrow P(\delta_{i+1})$, $Q \leftarrow Q(\delta_{i+1})$; $U \leftarrow U(\delta_{i+1})$; go to Step 3.

Step 2: $\{\delta_i > \delta_{i+1}\}$
Solve for δ, $(\delta - a_i)/b_i = p_i - p_{i+1}$.
$\delta_i \leftarrow \delta$, $P(\delta_i) \leftarrow a_i/b_i$, $Q(\delta_i) \leftarrow 1/b_i$, $U(\delta_i) \leftarrow p_{i+1}$.
Add a_i to the sorted sequence S; add δ_i to the top of T; stop.

Step 3: $\{\delta_i < \delta_{i+1}\}$
If $a_i + b_i u_i < \delta_{i+1}$ then add a_i and $a_i + b_i u_i$ to the sorted sequence S;
$U \leftarrow U(\delta_{i+1}) + u_i$.
Else, add a_i to S; $P \leftarrow P(\delta_{i+1}) + a_i/b_i$, $Q \leftarrow Q(\delta_{i+1}) + 1/b_i$.
Let the largest breakpoint lower than δ_{i+1} be v.
If v an a_j-breakpoint then go to Step 3(a).
If v an $(a_j + b_j u_j)$-breakpoint then go to Step 3(b).
If v a δ_i-breakpoint then go to Step 3(c).

Step 3(a): Let the breakpoint be a_k, set $\delta' = a_k$.
If $\delta' Q - P + U < p_i$ then $\delta_i > \delta'$; go to Step 4.
If $\delta' Q - P + U > p_i$ then $\delta_i < \delta'$; remove a_k from top of S;
$P \leftarrow P - a_k/b_k$, $Q \leftarrow Q - 1/b_k$; go to Step 3.
If $\delta' Q - P + U = p_i$, then $\delta_i = \delta'$; stop.

Step 3(b): Let the breakpoint be $a_k + b_k u_k$, set $\delta' = a_k + b_k u_k$.
If $\delta' Q - P + U < p_i$ then $\delta_i > \delta'$; go to Step 4.
If $\delta' Q - P + U > p_i$ then $\delta_i < \delta'$; remove $a_k + b_k u_k$ from top of S;
$U \leftarrow U - u_k$; go to Step 3.
If $\delta' Q - P + U = p_i$, then $\delta_i = \delta'$; stop.

Step 3(c): Let the largest breakpoint be δ_k; set $\delta' = \delta_k$.
If $\delta' Q - P + p_k < p_i$ then $\delta_i > \delta'$; go to Step 4.

If $\delta' Q - P + p_k > p_i$ then $\delta_i < \delta'$;
$P \leftarrow P + P(\delta')$, $Q \leftarrow Q + Q(\delta')$, $U \leftarrow U - (\delta' Q(\delta') - P(\delta'))$;
remove δ_k from top of the sequence T; go to Step 3.
If $\delta' Q - P + p_k = p_i$ then $\delta_i = \delta'$; stop.

Step 4: $\{\delta_i > \delta'\}$ Solve for δ, $\delta Q - P + U = p_i$.
Set $\delta_i = \delta$, $P(\delta_i) = P$, $Q(\delta_i) = Q$, $U(\delta_i) = U$; stop.

Lemma 4.3. *Algorithm* **Nested** *is correct. With a given sorted sequence of* $\{a_1, \ldots, a_n; a_1 + b_1 u_1, \ldots, a_n + b_n u_n\}$, *its complexity is* $O(n)$.

Proof. The dominant operation in the algorithm is adding element a_i and $b_i u_i$ to the sorted sequence S in Step 3. Using a straightforward approach of binary search, this takes $O(\log n)$ comparisons. We adopt here the UNION-FIND algorithm of Gabow and Tarjan [11]. Each subsequence is viewed as a collection of intervals that contains NO elements of a_i or $a_i + b_i u_i$, with endpoints at elements of the subsequence. Alternatively, the sorted sequence on $\{a_1, \ldots, a_n; a_1 + b_1 u_1, \ldots, a_n + b_n u_n\}$ may be viewed as an ordered vector. A subsequence is a 0–1 vector of length n with 1 in position j if the jth element is included in the subsequence and 0 otherwise. The aim is to maintain this 0–1 vector with pointers from each entry containing a 1, to the next such entry. The set of 0's separating each pair of 1's is an interval (that could be empty).

In order to position correctly an added item, we need to find an endpoint to the head (and tail) of the interval of elements to which it belongs. Since we have a given linear ordering of intervals the UNION-FIND algorithm applies. The other operation is SPLIT rather than UNION. Here when an element is added, an interval is split into two subsets. Still, an analogous algorithm to UNION can execute a sequence of p SPLIT-FIND operations on $2n$ elements in $O(2n + p)$ steps. In our case $p = 2n$, so the running time is linear.

Step 1 of the algorithm involves only a constant number of operations. If the outcome is to go to Step 2, δ is above all other breakpoints, then there is only $O(1)$ work. If however the outcome is that δ is below some of the breakpoints we may need to inspect several breakpoints, say q, prior to Step 4. In this case the amount of work in Step 3 is $O(q)$ except the adding operations.

The key observation is that, in the linear search for the Lagrange multiplier δ_i on the current sorted list S of breakpoints, once a breakpoint turns out to be larger than δ_i then the breakpoint is permanently deleted from the sequence and hence is not further considered in search for $\delta_{i-1}, \ldots, \delta_1$. This is, as mentioned earlier, because whenever δ is under certain breakpoints, the values of the variables at δ are upper bounds on the values of any optimal solution (Lemma 4.2). There is therefore no need to further consider any breakpoint above δ.

To summarize, if the search for δ goes up, as in Step 2 or 4, we add at most one breakpoint to the sequence, whereas if it goes down q breakpoints, then $q - 1$ breakpoints get deleted. Each call to Procedure Nested(i) creates at most three break-

points, a_k-type, $(a_k + b_k u_k)$-type, and δ_i. Let call i involve the inspection of q_i breakpoints. Then $\sum_{i=1}^{n} q_i = 3n + \sum_{i=1}^{n} 1$. Hence, the total number of operations is $O(n)$. □

5. Two strongly polynomial algorithms for (*Tree*)

In this section, we develop two strongly polynomial algorithms for the tree resource allocation problem which are more efficient than existing algorithms. The complexity of the first algorithm is $O(dn)$, where n is the number of variables and d is the depth of the underlying tree (see Fig. 1). If the tree is balanced, that is, $d = O(\log n)$, then the total complexity is $O(n \log n)$. The second algorithm runs in time $O(n \log n)$ and hence dominates the first. The second algorithm makes use of Algorithm Nested. The reason for presenting also the first algorithm is that it is simpler in structure and follows immediately from the properties of the solution on subtrees.

Two previously known strongly polynomial time algorithms are available for the problem (*Tree*). One is Tamir's algorithm [23] which minimizes the separable convex quadratic objective function on the feasible flows of a series-parallel network with single source and single sink. The algorithm complexity is $O(|A| \cdot |V| + |A| \log |A|)$ for the general problems where, $|A|$ is the number of arcs and $|V|$ is the number of nodes of the series-parallel network. For (*Tree*) it runs in $O(n^2)$ time (where n is the number of variables). The other algorithm follows from a result of Fujishige [9]. Fujishige devised an algorithm for the network resource allocation problem. The running time of this algorithm is dominated by the time required to solve at most $2|V| - 1$ maximum flow problems on the underlying network. The maximum flow problem on a tree is solvable in linear time. Hence Fujishige's algorithm, when applied to (*Tree*), also runs in $O(n^2)$ time.

Our algorithms rely on the recursive optimality structure of (*Tree*): the optimal solutions on subtrees are valid upper bounds of the optimal solution of the original problem. This property is established in Subsection 5.1. Subsections 5.2 and 5.3 include the description of the algorithms.

5.1. Optimality properties of (Tree)

Consider the tree resource allocation problem, (*Tree*), defined in Subsection 2.1. For notational convenience, we denote $S_0 = E$ $(= \{1, 2, \ldots, n\})$, $M = \{0, 1, \ldots, m\}$ and $p_0 = B$. Throughout this subsection, we assume that if $S_i \cap S_{i'} \neq \emptyset$ and $i < i'$ then $S_i \supseteq S_{i'}$. Allowing S_i's to be singleton sets, the problem can be rewritten as:

$$(Tree) \qquad \min \quad \sum_{j \in S_0} a_j x_j + \tfrac{1}{2} b_j x_j^2$$

$$\sum_{j \in S_i} x_j \leqslant p_i, \quad i \in M$$

$$x_j \geqslant 0, \quad j \in S_0.$$

Let λ_i be the Lagrange multiplier of the constraint on the index set S_i, $\sum_{j\in S_i} x_j \leqslant p_i$ and let $\sigma_j = \sum_{i\in I(j)}\lambda_i$, where $I(j) = \{i \in M \mid j \in S_i\}$. The Kuhn–Tucker optimality conditions for this case, referred to hereafter as (KT), are:

$$
(KT)\qquad
\begin{aligned}
&\text{(i)} && \lambda_i < 0 \quad \Rightarrow \quad \sum_{j\in S_i} x_j = p_i, \quad i \in M\\
&\text{(ii)} && x_j > 0 \quad \Rightarrow \quad a_j + b_j x_j - \sigma_j = 0, \quad j \in S_0\\
&\text{(iii)} && a_j + b_j x_j - \sigma_j \geqslant 0, \quad j \in S_0\\
&\text{(iv)} && \lambda_i \leqslant 0, \quad i \in M\\
&\text{(v)} && \sum_{j\in S_i} x_j \leqslant p_i, \quad i \in M\\
&\text{(vi)} && x_j \geqslant 0, \quad j \in S_0.
\end{aligned}
$$

Let $(Tree_i)$ be the tree resource allocation problem defined on a subtree rooted at node i of the underlying tree. For instance, $(Tree_0)$ is the problem $(Tree)$. Let C_i be the set of children of node i in the underlying tree. In particular, C_0 is the set of children of the root node 0 and $C_0 = \{1, 2, \ldots, l\}$. Then $(Tree_k)$, $k = 1, \ldots, l$, are resource allocation tree problems defined on the subtrees rooted at each child of the root node 0, and S_k is the index set of variables in $(Tree_k)$, (i.e., the leaves of $Tree_k$). Let M_k be the set of nodes in $(Tree_k)$. Then M_k are mutually disjoint and $M - \{0\} = M_1 \cup M_2 \cup \cdots \cup M_l$. Each problem $(Tree_k)$ can be written as:

$$
(Tree_k)\qquad
\begin{aligned}
\min \quad & \sum_{j\in S_k} a_j x_j + \tfrac{1}{2} b_j x_j^2\\
& \sum_{j\in S_i} x_j \leqslant p_i, \quad i \in M_k\\
& x_j \geqslant 0, \quad j \in S_k.
\end{aligned}
$$

For $k = 1, 2, \ldots, l$, let $\{\bar{x}_j \mid j \in S_k\}$ and $\{\bar{\lambda}_i \mid i \in M_k\}$ be the optimal solution and the set of optimal multipliers of $(Tree_k)$ respectively.

From (KT) applied to $(Tree_k)$, for every $j \in S_k$, we have

$$
\bar{x}_j > 0 \quad \Leftrightarrow \quad \bar{x}_j = \left(\bar{\sigma}_j - a_j\right)/b_j \quad \Leftrightarrow \quad \bar{\sigma}_j > a_j, \tag{5.1}
$$

where, $\bar{\sigma}_j = \sum_{i\in I_k(j)}\bar{\lambda}_i$ and $I_k(j) = \{i \in M_k \mid j \in S_i\}$.

When $\sum_{j\in S_0}\bar{x}_j \leqslant p_0$, then $\{\bar{x}_j \mid j \in S_0\}$ is an optimal solution of the original problem $(Tree)$. This follows since $\{\bar{x}_j \mid j \in S_0\}$ satisfies (KT) with multipliers $\lambda_0 = 0$ and $\lambda_i = \bar{\lambda}_i$ for $i \in M - \{0\}$.

On the other hand, when $\sum_{j\in S_0}\bar{x}_j > p_0$, then the solution $\{x_j(\delta) \mid j \in S_0\}$, defined in terms of the nonpositive parameter δ as follows, is feasible as stated in Lemma 5.1, and satisfies (KT):

If $\bar{x}_j = 0$ or equivalently $\bar{\sigma}_j \leqslant a_j$ (see (5.1)) then let $x_j(\delta) = 0$ for all $\delta \leqslant 0$. Otherwise, define

$$x_j(\delta) = \begin{cases} \bar{x}_j & \text{if } \delta \geqslant \bar{\sigma}_j, \\ \bar{x}_j - (\bar{\sigma}_j - \delta)/b_j & \text{if } \bar{\sigma}_j > \delta > a_j, \\ 0 & \text{if } a_j \geqslant \delta. \end{cases}$$

By (5.1), if $\bar{\sigma}_j > \delta > a_j$ in the above definition then

$$x_j(\delta) = \bar{x}_j - (\bar{\sigma}_j - \delta)/b_j = (a_j - \delta)/b_j. \tag{5.2}$$

Lemma 5.1. *For any fixed $\delta \leqslant 0$, the solution $\{x_j(\delta) \mid j \in E\}$ satisfies (KT) except possibly the constraint $\sum_{j \in S_0} x_j \leqslant p_0$.*

Proof. First let $\lambda_0 = \delta$. For $i \in M - \{0\}$, define parametrized multipliers $\lambda_i(\delta)$ in the following manner: Suppose that x_j is a variable of $(Tree_k)$ and $I_k(j) = \{i \in M_k \mid j \in S_i\} = \{i_1, i_2, \ldots, i_t\}$ with $i_1 < i_2 < \cdots < i_t$.

If $\delta < \bar{\sigma}_j$, then we let $\lambda_i(\delta) = 0$ for all $i \in I_k(j)$. If $\delta \geqslant \bar{\sigma}_j$, then find the minimum r such that $\bar{\lambda}_{i_1} + \bar{\lambda}_{i_2} + \cdots + \bar{\lambda}_{i_r} \leqslant \delta$, and for each $s = 1, \ldots, t$ set

$$\lambda_{i_s}(\delta) = \begin{cases} 0 & \text{if } s < r, \\ \bar{\lambda}_{i_1} + \bar{\lambda}_{i_2} + \cdots + \bar{\lambda}_{i_r} - \delta & \text{if } s = r, \\ \bar{\lambda}_{i_s} & \text{if } s > r. \end{cases}$$

First we need to verify that $\{\lambda_i(\delta) \mid i \in M_k\}$ are well-defined, i.e. if $s \in I_k(j) \cap I_k(j')$ with $j \neq j'$ then $\lambda_s(\delta)$ is uniquely determined. It was assumed that the sets S_i are indexed in such a way that $i < i'$ and $S_i \cap S_{i'} \neq \emptyset$ only if $S_i \supseteq S_{i'}$. So if $s \in I(j) \cap I(j')$ then $\{i \leqslant s \mid i \in M_k\} \cap I_k(j) = \{i \leqslant s \mid i \in M_k\} \cap I_k(j')$; hence the definition above uniquely determines $\lambda_{i_s}(\delta)$ for all $s \in M_k$.

Next we verify (KT). Since $0 \leqslant x_j(\delta) \leqslant \bar{x}_j$ for each $j \in S_0$ and $\bar{\lambda}_i \leqslant \lambda_i(\delta) \leqslant 0$ for each $i \in M - \{0\}$, these satisfy (iii), (iv) and (v) of (KT) except possibly the constraint $\sum_{j \in S_0} x_j \leqslant p_0$. Thus it remains to verify the complementary slackness conditions (i) and (ii),

(i) $\lambda_i(\delta) < 0 \Rightarrow \sum_{j \in S_i} x_j(\delta) = p_i$ for each $i \in M - \{0\}$,

(ii) $x_i(\delta) > 0 \Rightarrow \sum_{i \in I(j)} \lambda_i(\delta) + \lambda_0 = a_j + b_j x_j(\delta)$ for each $j \in E$.

To prove (i), assume for some $i' \in M - \{0\}$ we have $\sum_{j \in S_{i'}} x_j(\delta) < p_{i'}$. Either $\sum_{j \in S_{i'}} \bar{x}_j < p_{i'}$ or $\sum_{j \in S_{i'}} \bar{x}_j = p_{i'}$. In the former case, the optimality of $\bar{x}_j$ in the subproblem implies $\bar{\lambda}_{i'} = 0$. But, $0 \geqslant \lambda_i(\delta) \geqslant \bar{\lambda}_i$ for all i, hence $\lambda_{i'}(\delta) = 0$ as required.

In the latter case, since $0 \leqslant x_j(\delta) \leqslant \bar{x}_j$ for all j, the assumption implies that there exists $j' \in S_{i'}$ such that $0 \leqslant x_{j'}(\delta) < \bar{x}_{j'}$. So by the definition of $x_{j'}(\delta)$, $\bar{\sigma}_{j'} > \delta$. Then $\lambda_i(\delta) = 0$ for all $i \in I(j')$ by definition. Since $i' \in I(j')$, $\lambda_{i'}(\delta) = 0$.

To prove (ii), assume $x_{j'}(\delta) > 0$. Either $0 < x_{j'}(\delta) < \bar{x}_{j'}$ or, $0 < x_{j'}(\delta) = \bar{x}_{j'}$. In the former case, it follows from the definition of $x_{j'}(\delta)$ that $a_{j'} < \delta < \bar{\sigma}_{j'}$. Therefore, by definition of the parametric multipliers, $\lambda_i(\delta) = 0$ for all $i \in I(j')$ and hence $\delta = \sum_{i \in I(j')} \lambda_i(\delta) + \delta$. We set $\lambda_0 = \delta$. So it follows that $\delta = \sum_{i \in I(j')} \lambda_i(\delta) + \lambda_0$. Combin-

ing this with (5.2) which implies $\delta = a_{j'} + b_{j'} x_{j'}(\delta)$, we get $\sum_{i \in I(j')} \lambda_i(\delta) + \lambda_0 = a_{j'} + b_{j'} x_{j'}(\delta)$, as required.

In the latter case, when $0 < x_{j'}(\delta) = \bar{x}_{j'}$, $\delta \geqslant \bar{\sigma}_{j'}$ by the definition of $x_{j'}(\delta)$. From the definition of $\lambda_i(\delta)$, we have $\bar{\sigma}_{j'} = \sum_{i \in I(j')} \lambda_i(\delta) + \delta = \sum_{i \in I(j')} \lambda_i(\delta) + \lambda_0$. Combining this with (5.1), which is equivalent to $\bar{\sigma}_{j'} = a_{j'} + b_{j'} \bar{x}_{j'} = a_{j'} + b_{j'} x_{j'}(\delta)$, implies the statement of the lemma. □

In the above proof, $x_j(\delta) = \bar{x}_j$ when $\delta = 0$. As δ gets smaller below zero, $x_j(\delta)$ decreases piecewise linearly; $x_j(\delta)$ is either 0 or a piecewise linear function with two breakpoints a_j and $\bar{\sigma}_j$ which is constant outside the interval determined by the two points. Hence, $\sum_{j \in S_0} x_j(\delta)$ is a monotone increasing piecewise linear function when $\delta \leqslant 0$. Thus, by Lemma 5.1, the optimal solution of the problem (*Tree*) is either equal to $\{x_j(0) \mid j \in S_0\}$ in the case $\sum_{j \in S_0} \bar{x}_j \leqslant p_0$, or $\{x_j(\delta^*) \mid j \in S_0\}$ for $\delta^* < 0$ such that $\sum_{j \in S_0} x_j(\delta^*) = p_0$. So the optimal solution $\{x_j^* \mid j \in S_0\}$ of the original problem is bounded by the optimal solution $\{\bar{x}_j \mid j \in S_0\}$ of the subproblems, i.e. $x_j^* \leqslant \bar{x}_j$ for each $j \in S_0$.

For any nonnegative vector $\mathbf{x} = \{x_j \mid j \in S_k\}$ such that $x_j \leqslant \bar{x}_j$, $j \in S_k$, $\mathbf{x}$ is a feasible solution of the subproblem ($Tree_k$) for $k = 1, \ldots, l$. So we have the following corollary.

Corollary 5.2. *The optimal solution $\{\bar{x}_j \mid j \in S_k\}$ of the subproblem ($Tree_k$) provides valid upper bounds on the optimal solution of the problem (Tree). That is, we can replace the constraints in the subproblem ($Tree_k$) by the upper bound constraints*

$$x_j \leqslant \bar{x}_j \quad \textit{for all } j \in S_k,$$

without changing the optimal solution of the problem (Tree).

Corollary 5.2 is the key idea of the algorithms described in the following subsections. The next lemma establishes another useful feature that for each $i \in M$ the optimal solution of ($Tree_i$) satisfies the constraint $\sum_{j \in S_i x_j \leqslant p_i}$ with equality.

Lemma 5.3. *If the values of u_j and P_j are updated from the leaf nodes to the root:*

$$u_j \leftarrow \min\{u_j, \max\{-a_j/b_j, 0\}\} \quad \textit{and} \quad p_i \leftarrow \min\left\{p_i, \sum_{k \in C_i} p_k\right\}$$

(where C_i is the set of children of node i in the underlying tree), then the optimal solution of ($Tree_i$) satisfies the constraint on S_i, $\sum_{j \in S_i} x_j \leqslant p_i$, with equality for every $i \in M$.

Proof. The optimal value of each variable x_j is in the range where the derivative of the corresponding function is nonpositive. So it may be assumed that $u_j \leqslant \max\{-b_j/a_j, 0\}$, otherwise we can set $u_j \leftarrow \max\{-b_j/a_j, 0\}$ for each $j \in S_0$ without changing the

optimal solution. Also we may assume that $p_i \leqslant \sum_{k \in C_i} p_k$ for every i; otherwise the constraint on S_i would be redundant. The condition is satisfied for each node of (*Tree*) by setting $p_j \leftarrow \min\{p_i, \sum_{k \in C_i} p_k\}$, in $O(n)$ time.

It is left to show that $\sum_{j \in S_i} x_j \leqslant p_i$ is satisfied as equality in the optimal solution of ($Tree_i$) for all i. Suppose not, then let $\{\bar{x}_j \mid j \in S_i\}$ be the optimal solution of ($Tree_i$) for some $i \in M$ with $\sum_{j \in S_i} \bar{x}_j < p_i$. Since $p_i \leqslant \sum_{k \in C_i} p_k$, there is $k \in C_i$ such that $\sum_{j \in S_k} \bar{x}_j < p_k$. Repeating this, we find a path from node i to a leaf node representing an upper bound constraint problem $x_j \leqslant u_j$. In this path, the constraint corresponding to each node has positive slack with respect to the optimal solution $\{\bar{x}_j \mid j \in S_i\}$. Thus we can increase $\bar{x}_j$ by the smallest slack of the constraints in the path and strictly improve the objective value of ($Tree_i$). This contradicts the optimality of $\{\bar{x}_j \mid j \in S_i\}$. □

5.2. An O(*dn*) algorithm

Consider the problem (*Tree*) defined on a tree of depth d. When all p_i's are set as in Lemma 5.3, the optimal solution of ($Tree_i$) satisfies the constraint on S_i with equality. The subproblems defined on the subtrees rooted at the nodes of depth $d-1$ are (*SRA*)s or single variable problems with upper bounds (see Fig. 2). Call these (*SRA*)s the (*SRA*)*s at level d* of the problem (*Tree*). Let the (*SRA*)s at level d be ($Tree_{i_1}$), ..., ($Tree_{i_p}$) and the optimal solutions respectively $\{\bar{x}_j \mid j \in S_{i_k}\}$ for $k = 1, \ldots, p$. By repeated applications of Corollary 5.2, for each $k = 1, \ldots, p$ the optimal solution $\{\bar{x}_j \mid j \in S_{i_k}\}$ of ($Tree_{i_k}$) provides valid upper bounds on the optimal value of $\{x_j \mid j \in S_{i_k}\}$ in (*Tree*). So we can replace the constraints of ($Tree_{i_k}$) by upper bound constraints $\{x_j \leqslant \bar{x}_j \mid j \in S_{i_k}\}$. Thus after solving the (*SRA*)s at level d, we get an equivalent tree resource allocation problem of reduced depth, $d-1$. This procedure is repeated until we get the tree resource allocation problem of depth 1, which is an (*SRA*). Then the optimal solution to this (*SRA*) is the optimal solution to (*Tree*). The algorithm is formally presented as follows:

Procedure Depth

Step 0(a): {preprocess} From the leaf nodes to the root set $u_j \leftarrow \min\{u_j, \max\{-a_j/b_j, 0\}\}$ and $p_i \leftarrow \min\{p_i, \sum_{k \in C_i} p_k\}$.

Step 0(b): If (*Tree*) is a problem with single variable x_j (with $d = 0$) then stop. The optimal solution is $x_j = u_j$. Otherwise $l \leftarrow d$.

Step 1: Let the (*SRA*)s at level l be ($Tree_{i_1}$), ..., ($Tree_{i_p}$). For $k = 1, \ldots, p$, solve ($Tree_{i_k}$) by Procedure **SRA** and let the solution be $\{\bar{x}_j \mid j \in S_{i_k}\}$.

Step 2: If $l = 1$ then output the solution as the optimal solution; stop. Otherwise, continue.

Step 3: For $k = 1, \ldots, p$, update the upper bounds $u_j \leftarrow \bar{x}_j$ for all $j \in S_{i_k}$ and delete the constraints of ($Tree_{i_k}$) from (*Tree*).

Step 4: Set $l \leftarrow l - 1$. Go to Step 1.

For each $l = 1, \ldots, d$, the running time is dominated by the calls to Procedure **SRA** for solving the (*SRA*)s at level l. Since the total number of variables in the (*SRA*)s at any level l is bounded by n, this can be done in $O(n)$ time. Hence, the total complexity of Procedure **Depth** is $O(dn)$.

5.3. An $O(n \log n)$ algorithm

When Procedure **Depth** is applied to a (*Tree*) problem such as (*Nested*), then the running time is $O(n^2)$. Yet (*Nested*) can be solved in $O(n \log n)$ time. The second algorithm makes use of Algorithm **Nested**. The array $\{a_1, \ldots, a_n\}$ is initially sorted once. This sorted vector is then used in the linear time calls to Algorithm **Nested**.

The idea of the algorithm is inspired by the one used by Dyer and Frieze [7] for the convex tree allocation problem. That algorithm first finds a "long" path in the tree. Then it recursively finds optimal solutions on the subtrees rooted at nodes which are not on the path but have parents on the path. By Corollary 5.2, these optimal solutions provide valid upper bounds on the variables and thus reduce the tree resource allocation problem into an equivalent nested resource allocation problem. Finally the algorithm solves the nested resource allocation problem using the linear time algorithm (the sorting is given), Algorithm **Nested**.

In order to find a "long" path, for each node i of the tree we evaluate the number of nodes in the subtree rooted at node i, n_i. This can be done in $O(n)$ time by a simple dynamic programming procedure. Starting at the root of the tree as the initial node, the algorithm finds recursively a child node k of the current node i with $n_k > \frac{1}{2}n_i$. This is repeated until the current node is a leaf node. It is shown that the path obtained by this procedure is sufficiently "long".

Procedure Tree ((*Tree*))

Input: A tree resource allocation problem, (*Tree*).
Output: The optimal solution of (*Tree*).

Step 0(a): {preprocess} Sort $\{a_1, \ldots, a_n\}$ in increasing order.

Step 0(b): {preprocess} From the leaf nodes to the top node, set $u_j \leftarrow \min\{u_j, \max\{-a_j/b_j, 0\}\}$ and $p_i \leftarrow \min\{p_i, \Sigma_{k \in C_i} p_k\}$.

Step 0(c): If the problem has single variable x_j, terminate with the optimal solution $x_j = u_j$. Otherwise, let $i \leftarrow$ the root of tree.

Step 1: If i has a child k with $n_k > \frac{1}{2}n_i$ then $i \leftarrow k$ and repeat Step 1. Otherwise continue to Step 2.

Step 2: Let $P(i)$ be the path from the root to i. Define $K = \{k \notin P(i) \mid k$ has the parent in $P(i)\}$.

Step 3: For $k \in K$, call Procedure **Tree** (($Tree_k$)) and let $\{\bar{x}_j \mid j \in S_k\}$ be the optimal solution.
For $k \in K$, set $u_j \leftarrow \bar{x}_j$ for all $j \in S_k$.

Step 4: Let (*Nested*) be the nested problem defined by the constraints corresponding to the nodes in $P(i)$ and the updated upper bounds in Step 3.
Solve (*Nested*) by Algorithm **Nested**.

Theorem 5.4. *Procedure* **Tree** *is correct and solves the problem in* $O(n \log n)$ *steps.*

Proof. Step 1 finds the "long" path in tree in $O(n)$ time. Step 2 identifies the roots of subproblems appended in the path, which also can be done in $O(n)$ time. Step 3 solves the subproblems by recursive calls to Procedure **Tree**. Denote by $C(n)$ the complexity of Procedure **Tree** applied to the problem with n variables, then the total running time is $\sum_{k \in K} C(n_k)$ time. $O(n)$ time is required to update all upper bounds in Step 3. After Step 3, we obtain a nested resource allocation problem on n variables. Finally, Algorithm **Nested** solves this nested problem in $O(n)$ time using the presorted data.

The validity of the algorithm follows from Corollary 5.2 which ensures the equivalence of the original tree resource allocation problem and the nested resource allocation problem obtained in Step 3. Since every step except the recursive calls can be done in linear time, the total complexity is given by,

$$C(n) \leqslant \sum_{k \in K} C(n_k) + An, \tag{5.3}$$

for some constant A. Assume inductively that $C(m) \leqslant Dm \log m$ for all $m < n$ for some constant D. From (5.3),

$$C(n) \leqslant \sum_{k \in K} Dn_k \log n_k + An \leqslant D \log \tfrac{1}{2}n \sum_{k \in K} n_k + An = Dn \log \tfrac{1}{2}n + An.$$

Taking $D \geqslant A/\log 2$, we get $C(n) \leqslant Dn \log n$. By induction the stated complexity follows. □

6. A strongly polynomial algorithm for (*Network*)

Fujishige [9] showed how to solve for a *lexicographically optimal base* of a polymatroid using n calls to an oracle identifying a maximal independent vector of a polymatroid. Fujishige notes explicitly, that this algorithm is applicable for solving the problem (*Network*) with strictly convex and homogeneous (that is, without the linear terms $-a_k = 0$ for all k) cost function. As is easily established, the same algorithm applies with a minor modification also to the nonhomogeneous case, including linear terms, for a strictly convex cost function. Such algorithm requires in this case n calls to a procedure solving the maximum flow problem.

Gallo, Grigoriadis and Tarjan [12], in their significant work on parametric maximum flow problem, noted that their algorithm is applicable to the *lexicographically optimal flow problem*, and the problem is solvable in the running time of a single application of the preflow algorithm of Goldberg and Tarjan [13]. The lexicographically optimal flow also provides an immediate solution to (*Network*) if the cost function is strictly convex

and homogeneous. Unlike Fujishige's algorithm, the lexicographically optimal flow algorithm of [12] does not extend to the nonhomogeneous case without change in the running time. This is because, as explained later, their algorithm requires that the parametric capacities are linear in the parameter, while for (*Network*) with nonhomogeneous cost these capacities are piecewise linear. The purpose of this section is to devise and validate a lexicographically optimal flow algorithm that runs in the same time as a single application of the preflow algorithm, in the presence of piecewise linear parametric capacities. This algorithm is shown to be applicable to solving (*Network*) with nonstrictly convex and nonhomogeneous cost in the running time of a single application of the preflow algorithm.

It is also shown that the algorithm of [12] for finding all breakpoints of the cut capacity of a parametric flow network with linear parametric capacities has invalid initialization procedure. We propose an alternative valid initialization procedure that corrects for the flaw in that algorithm.

The equivalence of (*Network*) and the lexicographically optimal flow problem is discussed in Subsection 6.1. In Subsection 6.2, we show how to formulate a *parametric flow problem* to solve the lexicographically optimal flow problem. Subsections 6.3 and 6.4 contain the properties of the parametric flow problem which are used to prove the validity of the algorithm presented in Subsections 6.5 and 6.6. In Subsections 6.5 and 6.6, we present the algorithm, based on the algorithms in [12], that solves the lexicographically optimal flow problem.

6.1. Lexicographically optimal flow problem

Let G be the multiple sink flow network on which problem (*Network*) is defined (as in Subsection 2.1, (5)). The following observations and assumptions simplify the problem: The optimal solution $\mathbf{x}$ satisfies $x_k \leqslant -a_k/b_k$, hence we may set $u_k \leftarrow \min\{u_k, -a_k/b_k\}$; As the maximum flow value may not exceed the capacities of the arcs adjacent to the sink, $B \leftarrow \min\{B, \sum_{t_k \in T} u_k\}$, and it is assumed without loss of generality that B is the maximum flow value. To ensure that, we add an additional arc going into the sink s with capacity equal to B and cost zero. The flow on that arc is then one component of the out-flow vector.

Let the set of sinks be $T = \{t_1, t_2, \ldots, t_n\}$ and x_i the flow from sink t_i to t. Then the network resource allocation problem is rewritten as:

$$(\textit{Network}) \qquad \min \quad \sum_{k=1}^{n} \tfrac{1}{2} b_k x_k^2 + a_k x_k$$

$$\text{s.t.} \quad \mathbf{x} = (x_1, x_2, \ldots, x_n) \text{ is the out-flow vector of a maximum flow } \mathbf{f} \text{ of } G.$$

The corresponding *lexicographically optimal flow problem* is defined:

(*Lexico*) Find a maximum flow $\mathbf{f}$ which lexicographically maximizes the n-component vector whose kth component is the kth smallest element of $\{b_k x_k + a_k \mid t_k \in T\}$.

The following theorem due to Fujishige [10] and an additional observation stated as Lemma 6.2 establishes the equivalence of (*Network*) and (*Lexico*).

Theorem 6.1. *Let r be a submodular function defined on a distributive lattice Λ of subsets of E, a finite set, and let $\{g_k(x_k) \mid k \in E\}$ be differentiable convex functions. Consider the following optimization problem (which is similar to (GAP) defined in Section 1 except that the constraint on the set E is equality and there are no nonnegativity constraints)*

$$(GAP') \qquad \min \quad \sum_{k\in E} g_k(x_k)$$
$$\sum_{k\in E} x_k = r(E)$$
$$\sum_{k\in A} x_k \leqslant r(A), \quad A \in \Lambda.$$

Then $\mathbf{x}$ *is an optimal solution of (GAP') if and only if* $\mathbf{x}$ *lexicographically maximizes the vector whose kth component is the kth smallest element of the vector of derivatives, $\{g_k'(x_k) \mid k \in E\}$.*

Lemma 6.2. *Let r be defined on the lattice of all subsets of E. Assume that r is monotone, i.e. for every pair of subsets $A, B \subseteq E$ with $A \subseteq B$ we have $r(A) \leqslant r(B)$. Then every feasible solution of (GAP') is nonnegative.*

Proof. Suppose that $\bar{\mathbf{x}}$ is a feasible solution of (*GAP'*) with $\bar{x}_p < 0$ for some $p \in E$. Then,

$$r(E - \{p\}) \geqslant \sum_{k\in E-\{p\}} \bar{x}_k = r(E) - \bar{x}_p > r(E) \geqslant r(E - \{p\}),$$

which is a contradiction. □

Theorem 6.1 has an analogue for the network resource allocation problem.

Theorem 6.3. $\mathbf{x}^* = \{x_k^* \mid t_k \in T\}$ *is an optimal solution of (Network) if and only if* $\mathbf{x}^*$ *is the out-flow vector of a solution* $\mathbf{f}$ *of (Lexico).*

Proof. For all $S \subseteq T$, define $r(S)$ to be the value of maximum flow achievable through the subset of sinks, S. In particular, $r(T) = B$, the maximum flow value of G. It is known (see e.g. [9]) that r is a submodular function and (*Network*) can be written as

$$(P(6.1)) \qquad \sum_{k=1}^{n} \tfrac{1}{2} b_k x_k^2 + a_k x_k$$
$$\text{(i)} \quad \sum_{k=1}^{n} x_k = B \ (= r(T))$$
$$\text{(ii)} \quad \sum_{t_k\in S} x_k \leqslant r(S), \quad S \subseteq T$$
$$\text{(iii)} \quad x_k \geqslant 0 \quad \text{for } k = 1, \ldots, n.$$

r is a monotone function defined on the set of all subsets of T. So by Lemma 6.2, the nonnegativity constraints, (iii) can be relaxed from (P(6.1)) without changing the solution. Thus our theorem directly follows from Theorem 6.1. □

For the case when $a_k = 0$ and $b_k > 0$ for all $k = 1, \ldots, n$, Fujishige [9] developed a strongly polynomial algorithm which solves (*Lexico*) in the time required to solve at most $2n - 1$ maximum flow problems on G. It can be easily shown that the same algorithm can be used to solve the general case in which $\mathbf{a} \neq 0$ in the same running time by the translation, $y_k = x_k - a_k/b_k$ of the submodular polyhedron of (P(6.1)) as the translation preserves the submodularity. This running time exceeds the running time established here by a factor of $O(n)$.

6.2. *The parametric flow problem*

In order to solve the problem (*Lexico*), we consider the network G with parametric capacities $c_k(\lambda)$ assigned to each arc (t_k, t) for $k = 1, \ldots, n$, with $c_k(\lambda) = \max\{0, (\lambda - a_k)/b_k\}$ defined for $\lambda \geqslant \min\{a_k \mid k = 1, \ldots, n\}$. As shown in the following subsection, from the breakpoints of the parametric flow problem defined on G with the parametric capacities, one can construct a solution of (*Lexico*) and hence a solution of (*Network*).

The parametric capacities functions $c_k(\lambda)$ are monotone increasing in λ, where the parametric algorithm of [12] which we use requires that the capacity functions at the sink are nonincreasing, and at the source they are nondecreasing. To this end we reintroduce the problem with the reversed roles of source and sinks.

(*Lexico'*) Find a maximum flow $\mathbf{f}$ on G which lexicographically maximizes the n-component vector whose kth element is the kth smallest element of $\{b_k x_k + a_k \mid s_k \in S\}$.

In the reversed network, $\hat{G}$, we have a sink s and arcs (s, s_k) for $k = 1, \ldots, n$ (each s_k corresponds to a t_k in G.) Each arc (s, s_k) of $\hat{G}$ is assigned the parametric capacity $c_k(\lambda)$. Denote this parametric flow network by $\hat{G}(\lambda)$; so $\hat{G}(\bar{\lambda})$ with some fixed value $\lambda = \bar{\lambda}$ stands for the network $\hat{G}$ with capacity $c_k(\bar{\lambda})$ on each arc (s, s_k) and $\hat{G}(\infty)$ is $\hat{G}$ with each arc (s, s_k) assigned infinite capacity. For an s–t cut $(X, \bar{X})$ of $\hat{G}$, $c_\lambda(X, \bar{X})$ denotes the capacity of $(X, \bar{X})$ in $\hat{G}(\lambda)$. Let $\kappa(\lambda)$ be the capacity of minimum s–t cut of $\hat{G}(\lambda)$.

In order to establish the validity of the algorithm several important properties of $\hat{G}(\lambda)$ are considered first.

6.3. *Properties of* $\hat{G}(\lambda)$

The minimum cut capacity function $\kappa(\lambda)$, as shown in Subsection 6.6, is a monotone nondecreasing piecewise linear function. Let the *breakpoint* of $\kappa(\lambda)$ be the value of λ where the slope of $\kappa(\lambda)$ changes. At certain breakpoints, some nodes of $\hat{G}$ shift from the sink side to source side as λ increases; we call such breakpoints *node-shifting* breakpoints.

For each $k = 1, \ldots, n$, let λ_k be the node-shifting breakpoint where s_k shifts from sink side to source side. Without loss of generality, we may assume that $\lambda_1 \leqslant \lambda_2 \leqslant \cdots \leqslant \lambda_n$. Let $k_1, k_2, \ldots, k_{p-1}$ be the values of k such that $\lambda_{k_j} < \lambda_{k_j+1}$. Let $k_0 = 0$ and $k_p = n$. Then $\{\lambda_{k_1}, \lambda_{k_2}, \ldots, \lambda_{k_p}\}$ is the subsequence of all distinct values of λ_k sorted in increasing order:

$$\lambda_1 = \cdots = \lambda_{k_1} < \lambda_{k_1+1} = \cdots = \lambda_{k_2} < \cdots < \lambda_{k_{j-1}+1} = \cdots = \lambda_{k_j} < \cdots < \lambda_{k_{p-1}+1} = \cdots = \lambda_{k_p}.$$

Lemma 6.4. (i) *For $j = 1, \ldots, p$, $(x_1, \ldots, x_{k_j}) = (c_1(\lambda_1), \ldots, c_{k_j}(\lambda_{k_j}))$ attains the maximum flow value achievable through sinks $\{s_1, \ldots, s_{k_j}\}$, i.e. $\sum_{k=1}^{k_j} x_k$ is the value of the maximum flow of the multiple source network G with the additional restrictions that $x_{k_j+1} = \cdots = x_n = 0$.*

(ii) *On $\hat{G}(\infty)$, there is a maximum flow* **f** *which gives the in-flow vector* **x** *such that $x_k = c_k(\lambda_k)$ for all $k = 1, \ldots, n$.*

Proof. The proof is essentially identical to the proof of Theorem 4.1 of [12] and is repeated here for completeness sake.

(i) Let $\{s\} = X_0 \subset X_1 \subset X_2 \subset \cdots \subset X_p$ be the sets such that $(X_j, \bar{X}_j)$ for $j = 1, \ldots p$ is the minimum cut with the largest source side of $\hat{G}(\lambda_{k_j})$. Then $s_{k_{j-1}+1}, s_{k_{j-1}+2}, \ldots, s_{k_j} \in X_j - X_{j-1}$. For $j = 1, \ldots, n$, the cut $(X_{j-1}, \bar{X}_{j-1})$ is a minimum cut for $\hat{G}(\lambda_{k_j})$ as well. Thus $c_{\lambda_{k_j}}(X_{j-1}, \bar{X}_{j-1}) = c_{\lambda_{k_j}}(X_j, \bar{X}_j)$; see Fig. 5. It follows by induction on $j = 1, \ldots, p$,

$$c_{\lambda_{k_j}}(X_j, \bar{X}_j) = \sum_{k=1}^{k_j} c_k(\lambda_k) + \sum_{k=k_j+1}^{n} c_k(\lambda_{k_j}), \tag{6.1}$$

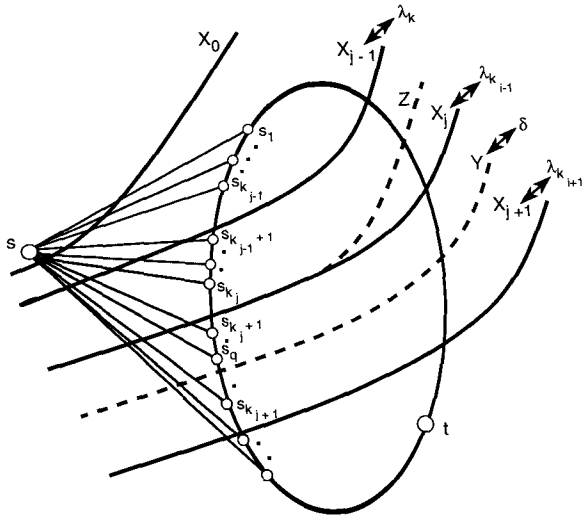

Fig. 5. $\hat{G}(\lambda)$.

which implies for $j = 1, \ldots, p$,

$$c_\infty\big(X_j - \{s\}, \bar{X}_j\big) = \sum_{k=1}^{k_j} c_k(\lambda_k). \tag{6.2}$$

Which, in turn, implies (i).

(ii) Consider the maximum flows, $\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_p$ generated by the parametric flow algorithm of [12] for the successive parameter λ values $\lambda_{k_1}, \lambda_{k_2}, \ldots, \lambda_{k_p}$. When the parametric maximum flow algorithm is restarted with new value λ_{k_j} of λ, the flow on each arc (s, s_k) with $k \in \{k_{j-1}+1, k_{j-1}+2, \ldots, n\}$ is first increased from $c_k(\lambda_{k_{j-1}})$ to $c_k(\lambda_{k_j})$. This additional flow will reach the sink t, because of (6.1) and the fact that $(X_j, \bar{X}_j)$ is the minimum cut of $\hat{G}(\lambda_{k_j})$. By repeating the argument inductively on $j = 1, \ldots, p$, we have $x_k = c_k(\lambda_k)$ for $k = 1, \ldots, n$. In particular $\mathbf{f}_p$ is the desired flow. □

Remark 1. The proof holds for any type of parametric capacities once they satisfy the monotonicity assumption and the range of λ begins at the point where all parametric capacities are zero.

Remark 2. In the proof of Theorem 4.1 of [12], the cut $(X_j, \bar{X}_j)$ is claimed to be not only the minimum s–t cut but also the smallest source side of minimum s–t cut of $\hat{G}(\lambda_{k_j+1})$. This is false even in the simpler case where the parametric capacities are linear functions without constant terms: if there is alternate cut $(Z, \bar{Z})$ such that $s_1, \ldots, s_{k_j} \in Z$, $c(Z - \{s\}, \bar{Z}) = c(X_j - \{s\}, \bar{X}_j)$ and Z is properly contained in X_j (see Fig. 5), then $(Z, \bar{Z})$ is also a minimum s–t cut of $\hat{G}(\lambda_{k_j+1})$, and such an example is not hard to construct. Consequently, the breakpoint algorithm as currently stated in [12] is invalid, since "contracted" subsets of vertices in the initialization step are not necessarily disjoint. In Subsection 6.4, we define a new method of "contracting" a pair of subsets of vertices which are not disjoint but still possess the property required for the breakpoint algorithm.

Lemma 6.5. *Let δ be a value such that $\lambda_{k_j} < \delta < \lambda_{k_{j+1}}$ for some $1 \leqslant j \leqslant p-1$. Then $(X_j, \bar{X}_j)$ is the minimum s–t cut with largest source side of $\hat{G}(\delta)$.*

Proof. Let $(Y, \bar{Y})$ be the minimum s–t cut of $\hat{G}(\delta)$ such that $|Y|$ is maximum. Then since $\lambda_{k_j} < \delta$, we have $X_j \subseteq Y$. Our claim is that $X_j = Y$. So assume the contrary: assume that Y properly contains X_j. Then Y must contain an element $s_q \in \{s_{k_j+1}, s_{k_j+2}, \ldots, s_{k_{j+1}}\}$ with $c_q(\delta) > 0$ (see Fig. 5) since otherwise

$$c\big(Y - \{s\}, \bar{Y}\big) = c\big(X_j - \{s\}, \bar{X}_j\big), \tag{6.3}$$

and hence $c_{\lambda_{k_j}}(Y, \bar{Y}) = c_{\lambda_{k_j}}(X_j, \bar{X}_j)$. This implies that $(Y, \bar{Y})$ is a minimum cut of $\hat{G}(\lambda_{k_j})$. But it contradicts the maximality of $|X_j|$.

This however means that δ is a breakpoint of $\kappa(\lambda)$ at which the node s_q shifts from the sink side to source side. But it is not possible since $\lambda_{k_{j+1}}$ is the smallest breakpoint larger than λ_{k_j} and $\lambda_{k_j} < \delta < \lambda_{k_{j+1}}$. So $X_j = Y$. □

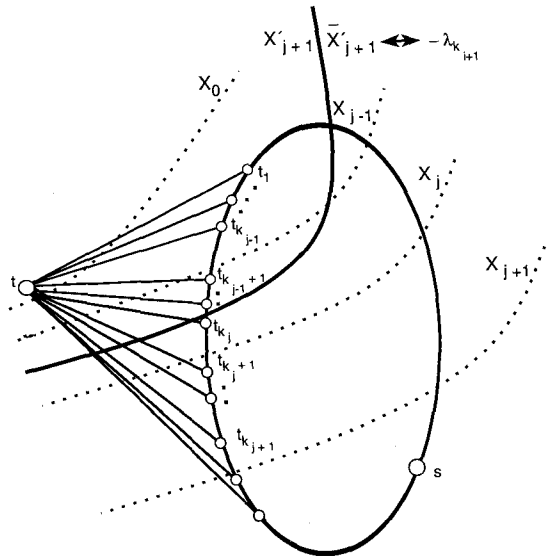

Fig. 6. $\hat{G}^{R}(-\mu)$.

Consider now the reversed parametric network $\hat{G}^{R}(\lambda)$ of $\hat{G}(\lambda)$, i.e. the parametric network obtained from $\hat{G}(\lambda)$ by reversing the directions of all arcs, considering the sources as sinks and sink as source while maintaining the same capacities (Fig. 6). When the parametric maximum flow algorithm of [12] is applied to $\hat{G}^{R}(\lambda)$, the parameter λ is replaced by $-\mu$ in order to satisfy the monotonicity requirement. For $j = p, p-1, \ldots, 1$, the algorithm finds the minimum cut with the largest source side $(\bar{X}'_j, X'_j)$ (in the reversed network) for the breakpoint $-\lambda_{k_j}$; see Fig. 6. for $-\lambda_{k_{j+1}}$. An argument analogous to the one in the proof of Lemma 6.5 implies that if $\lambda_{k_j} < \delta < \lambda_{k_{j+1}}$ (so $-\lambda_{k_{j+1}} < -\delta < -\lambda_{k_j}$), then $(\bar{X}'_{j+1}, X'_{j+1})$ is the minimum cut with the largest source side for $\mu = -\delta$ in the reversed network.

Corollary 6.6. *Let δ be a value such that $\lambda_{k_j} < \delta < \lambda_{k_{j+1}}$ for some $1 \leqslant j \leqslant p-1$. Then $(X'_{j+1}, \bar{X}'_{j+1})$ is the minimum s–t cut with smallest source side of $\hat{G}(\delta)$ where $(X'_j, \bar{X}'_j)$ denotes the minimum s–t cut with smallest source side of $\hat{G}(\lambda_{k_j})$ for $k = 1, 2, \ldots, n$.*

The following lemma is also needed for the algorithms in subsequent subsections.

Lemma 6.7. (i) $X'_{j+1} \subseteq X_j$ *but,* (ii) X'_{j+1} *is not contained in* X_{j-1}.

Proof. (i) follows from the fact that $(X_j, \bar{X}_j)$ is a minimum cut of $\hat{G}(\lambda_{k_{j+1}})$ as mentioned in the proof of Lemma 6.4.

To prove (ii), assume $X'_{j+1} \subseteq X_{j-1}$. Then in $\hat{G}(\lambda_{k_{j+1}})$, for every $k \in \{k_{j-1}+1, k_{j-1}+2, \ldots, k_j\}$, the arc (s, s_k) is saturated with the flow equal to $c_k(\lambda_{k_{j+1}})$. But $c_k(\lambda_{k_{j+1}}) \geqslant c_k(\lambda_{k_j})$ for every k. Furthermore, there is at least one index $q \in \{k_{j-1}+1, k_{j-1}+$

$2, \ldots, k_j\}$ such that $c_q(\lambda_{k_j}) > 0$ since otherwise $(X_j, \bar{X}_j)$ would not have been an s–t minimum cut of $\hat{G}(\lambda_{k_j})$. This implies that $c_q(\lambda_{k_{j+1}}) > c_q(\lambda_{k_j})$.

Thus (6.1) implies that for $I + \{1, 2, \ldots, k_{j-1}\} \cap (X'_{j+1} - \{s\})$,

$$c_\infty\left(X'_{j+1} - \{s\}, \bar{X}'_{j+1}\right) < \sum_{k \in I} c_k(\lambda_k).$$

But in $\hat{G}(\lambda_{k_{j-1}})$ for each $k \in I$, the flow $c_k(\lambda_k)$ on the arc (s, s_k) can reach the sink t through the cut $(X'_{j+1} - \{s\}, \bar{X}'_{j+1})$ of G. Hence,

$$c_\infty\left(X'_{j+1} - \{s\}, \bar{X}'_{j+1}\right) \geqslant \sum_{k \in I} c_k(\lambda_k),$$

which is a contradiction. □

6.4. The contraction of $\hat{G}(\lambda)$

Let δ and Δ be different values of λ with $\delta < \Delta$. Assume that all parametric capacities are linear functions of λ on the closed interval $[\delta, \Delta]$ (but not necessarily outside the interval). Let $\mathbf{f}(g)$ be a maximum flow and $(W, \overline{W})$ $((\overline{Z}, Z)$, respectively) be the corresponding minimum cut with the largest source side of $\hat{G}(\delta)$ $(\hat{G}^R(\Delta)$, respectively).

The purpose of this subsection is to show that the node-shifting breakpoints of $G(\lambda)$ on the open interval (δ, Δ) can be found by the breakpoint algorithm of [12]. We first present a modified initialization procedure to correct for the flaw addressed in Remark 2.

The initialization procedure of the algorithm *contracts* W and $\bar{Z}$ into source and sink respectively, where by the *contraction* of a subset of vertices we mean shrinking of the vertices of the set into a single vertex, eliminating loops and combining arcs by adding their capacities. This contraction procedure is to achieve the property that

(*) in the contracted network, the s–t cut with the trivial source side $\{s\}$ (sink side $\{t\}$) is the unique cut which corresponds to a minimum s–t cut of $\hat{G}(\delta)$ $(\hat{G}(\Delta)$, respectively),

where the correspondence is the one obtained by expanding the contracted vertex set.

However, as pointed out in the Remark 2 of the previous subsection, W and $\bar{Z}$ are not necessarily disjoint and the contraction procedure as proposed in [12] is invalid.

The following preliminary lemma is needed to establish the modified initialization procedure. Its proof follows directly from Lemma 6.5, Corollary 6.6 and Lemma 6.7.

Lemma 6.8. *$\hat{G}(\lambda)$ has node-shifting breakpoint on the open interval (δ, Δ) if and only if W does not include Z; or equivalently, $W \cup \bar{Z}$ is a proper subset of the vertex set of $\hat{G}(\lambda)$.*

Procedure Contraction$(\hat{G}(\lambda); W, \bar{Z})$

Step 1: For every source $s_q \in W \cap \bar{Z}$, delete the arc (s, s_q) from $\hat{G}(\lambda)$.

Step 2: Contract $W - \bar{Z}$ and $\bar{Z}$ into single vertices. Call this contracted parametric network $\hat{G}_{(\delta, \Delta)}(\lambda)$.

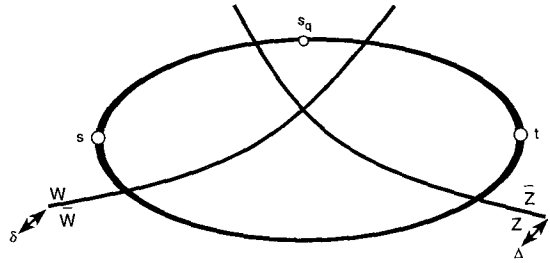

Fig. 7.

Let $\mathbf{f}'$ be the flow $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ which corresponds to $\mathbf{f}$ of $\hat{G}(\lambda)$.
Let $\mathbf{g}'$ be the flow of $\hat{G}^{\mathrm{R}}_{(\delta,\,\Delta)}(\lambda)$ which corresponds to $\mathbf{g}$ of $\hat{G}^{\mathrm{R}}(\lambda)$.

Step 3: If $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ has at least three vertices, continue to the (main procedure) of the breakpoint algorithm with initial values $\mathbf{f}'$, $\mathbf{g}'$, δ and Δ; otherwise $\hat{G}(\lambda)$ has no node-shifting breakpoints on (δ, Δ); stop.

Suppose a source $s_q \in W \cap \bar{Z}$ in Step 1 (see Fig. 7). Since $\delta < \Delta$, $c_s(W, \bar{W}) = c_\delta(W \cap Z, \overline{W \cap Z})$. So $x_q = 0$ in $\mathbf{f}$. Once s_q is in the source side of an s–t minimum cut of $\hat{G}(\delta)$, for every $\lambda \geqslant \delta$, there is an s–t minimum cut of $\hat{G}(\lambda)$ in which s_q is in the source side. Thus for every $\lambda \geqslant \delta$, $x_q = 0$ in the maximum flow of $\hat{G}(\lambda)$. So we can delete (s, s_q) for every $s_q \in W \cap \bar{Z}$ without changing the breakpoints.

Consider Step 2 of the contraction. W is maximum and $W \cap \bar{Z}$ is shrunk into the sink. Hence, in the contracted network, $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ where $W - \bar{Z}$ is shrunk into source, the s–t cut with the trivial source side is the unique cut corresponding to a minimum s–t cut of $\hat{G}(\delta)$ (see Fig. 7). Also by the maximality of $\bar{Z}$, it is clear that the s–t cut of $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ with the trivial sink side is the unique cut corresponding to an s–t minimum cut of $\hat{G}(\Delta)$. Thus $(*)$ is achieved in this contraction procedure.

By Lemma 6.8, $\hat{G}(\lambda)$ has node-shifting breakpoints on (δ, Δ) if and only if $W \cup \bar{Z}$ is a proper subset of the vertex set of $\hat{G}(\lambda)$; which means $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ has more than two vertices. Thus if $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ has only two vertices, source and sink, then we conclude that the original problem has no breakpoints in (δ, Δ) and terminate. Otherwise we apply the breakpoint algorithm of [12] to $\hat{G}_{(\delta,\,\Delta)}(\lambda)$.

The details of the breakpoint algorithm of [12] are now briefly sketched: The (main procedure of the) breakpoint algorithm of [12] starts with a pair of parametric flow networks: $\hat{G}_{(\delta,\,\Delta)}(\lambda)$ with initial preflow $\mathbf{f}'$ and $\hat{G}^{\mathrm{R}}_{(\delta,\,\Delta)}(\lambda)$ with initial preflow $\mathbf{g}'$. Under the assumption that all parametric capacities are linear on $[\delta, \Delta]$, the minimum cut capacity is a piecewise linear concave function of λ on $[\delta, \Delta]$. Thus the next guess η of a node-shifting breakpoint on (δ, Δ) can be calculated as the intersection of two tangent lines determined by the leftmost and the rightmost line segments of the (piecewise linear concave) minimum cut capacity function (restricted on $[\delta, \Delta]$). The algorithm determines whether η is a node-shifting breakpoint by calculating the tangent lines determined by the s–t minimum cuts with the largest source side and the smallest source side

for $\lambda = \eta$; if they do not coincide then η is indeed a node-shifting breakpoint. The algorithm repeats this procedure on the next search intervals $[\delta, \eta]$ and $[\eta, \Delta]$. To obtain efficient time bound, in the current interval $[\delta, \Delta]$ the algorithm finds a maximum flow (and a minimum cut) for $\lambda = \eta$ by concurrent invocation of the preflow algorithm, for both $\hat{G}_{(\delta, \Delta)}(\lambda)$ with initial preflow $\mathbf{f}'$ and $\hat{G}^{R}_{(\delta, \Delta)}(\lambda)$ with initial preflow $\mathbf{g}'$. By doing this the algorithm can "balance" the numbers of nodes between the source side and the sink side of the minimum cuts considered in the subsequent search intervals. The algorithm finds the node-shifting breakpoints on (δ, Δ) in $O(N'M' \log(N'^2/M'))$ steps, where M' and N' are respectively the numbers of arcs and nodes of $\hat{G}_{(\delta, \Delta)}(\lambda)$.

6.5. The main algorithm

When $\mathbf{a} = 0$, the main algorithm is identical to the algorithm in Subsection 4.1 of [12] with the exception of using the modified subroutine, Subroutine **Breakpoint-Finder** for finding the node-shifting breakpoints of the minimum cut capacity function $\kappa(\lambda)$, of $\hat{G}(\lambda)$.

Algorithm Lexico-Finder

Step 0: Augment the network G with a dummy source s and the arcs (s, s_k) for $k = 1, \ldots, n$.
Call this augmented network $\hat{G}$.

Step 1: Assign the parametric capacity $c_k(\lambda) = \min\{0, (\lambda - a_k)/b_k\}$ to each arc (s, s_k) of $\hat{G}$. Denote the parametric flow network by $\hat{G}(\lambda)$.

Step 2: Call Subroutine **Breakpoint-Finder** to find the node-shifting breakpoints of $\kappa(\lambda)$ for $\lambda \geqslant \min\{a_k \mid k = 1, \ldots, n\}$

Step 3: For each source s_k, $k = 1, \ldots, n$, find the node-shifting breakpoint λ_k at which s_k shifts from sink side to source side of a minimum s–t cut of $\hat{G}(\lambda_k)$ as λ increases.

Step 4: Assign the capacity $c_k(\lambda_k)$ to each arc (s, s_k) of $\hat{G}$.
Find a maximum flow $\mathbf{f}$ on the network with these upper bounds.
Output the in-flow vector $\mathbf{x}$ of $\mathbf{f}$ as the optimal solution of (*Lexico'*).

Subroutine **Breakpoint-Finder** is given in the next subsection. In the following theorem, we prove the validity of the main algorithm under the assumption that Subroutine **Breakpoint-Finder** correctly finds the node-shifting breakpoints of $\kappa(\lambda)$.

Theorem 6.9. *Algorithm* **Lexico-Finder** *is correct: the maximum flow* $\mathbf{f}$ *obtained in Step* 4 *gives the in-flow vector* $\mathbf{x}$ *which is the optimal solution of the problem* (*Lexico'*).

Proof. Consider the breakpoints of Step 3, $\lambda_1, \ldots, \lambda_n$. Without loss of generality, we may assume that $\lambda_1 \leqslant \lambda_2 \leqslant \cdots \leqslant \lambda_n$. Let $k_1, k_2, \ldots, k_{p-1}$ be the values of k such that

$\lambda_{k_j} < \lambda_{k_j+1}$. Let $k_0 = 0$ and $k_p = n$. Then $\{\lambda_{k_1}, \lambda_{k_2}, \ldots, \lambda_{k_p}\}$ is the subsequence of all distinct values sorted in increasing order: Also let $\mathbf{f}$ be the maximum flow obtained in Step 4.

By Lemma 6.4, the in-flow vector $\mathbf{x}$ of $\mathbf{f}$ satisfies $x_k = c_k(\lambda_k)$ for all $k = 1, \ldots, n$ and:

Fact 1. For $j = 1, \ldots, p$, $(x_1, \ldots, x_{k_j}) = (c_1(\lambda_1), \ldots, c_{k_j}(\lambda_{k_j}))$ attains the maximum flow value achievable through sinks $\{s_1, \ldots, s_{k_j}\}$, i.e. $\sum_{k=1}^{k_j} x_k$ is the value of the maximum flow of the multiple source network G with the additional restrictions that $x_{k_j+1} = \cdots x_n = 0$.

From the definition of $c_k(\lambda)$:

$$c_k(\lambda_k) = \begin{cases} (\lambda_k - a_k)/b_k & \text{if } \lambda_k \geqslant a_k, \\ 0 & \text{if } \lambda_k < a_k. \end{cases}$$

Define $\mu_k = \max\{\lambda_k, a_k\}$. Then we have:

Fact 2. $c_k(\lambda_k) = c_k(\mu_k) = (\mu_k - a_k)/b_k$ for all $k = 1, \ldots, n$.

Fact 3. $\{\lambda_k \mid k = 1, \ldots, n\}$ and $\{\mu_k \mid k = 1, \ldots, n\}$ have the identical elements except for k's such that $\lambda_k < \mu_k$ and $c_k(\lambda_k) = c_k(\mu_k) = 0$.

Let σ be the permutation of $\{1, \ldots, n\}$ such that $\mu_{\sigma(1)} \leqslant \mu_{\sigma(2)} \leqslant \cdots \leqslant \mu_{\sigma(n)}$. Let $i_1, i_2, \ldots, i_{q-1}$ be the values of i such that $\mu_{\sigma(i_j)} < \mu_{\sigma(i_j+1)}$. Let $i_0 = 0$ and $i_q = n$. Then $\{\mu_{\sigma(i_1)}, \mu_{\sigma(i_2)}, \ldots, \mu_{\sigma(i_q)}\}$ is the set of all the distinct values in increasing order:

$$\begin{aligned} \mu_{\sigma(1)} = \cdots = \mu_{\sigma(i_1)} < \mu_{\sigma(i_1+1)} = \cdots = \mu_{\sigma(i_2)} < \cdots < \mu_{\sigma(i_{j-1}+1)} \\ = \cdots = \mu_{\sigma(i_j)} < \cdots < \mu_{\sigma(i_{q-1}+1)} = \cdots = \mu_{\sigma(i_q)}. \end{aligned}$$

For a fixed $j = 1, \ldots, q$, let j^* denote the maximum value such that $\lambda_{k_{j^*}} \leqslant \mu_{\sigma(i_j)}$, then since $\lambda_k \leqslant \mu_k$ for all $k = 1, \ldots, n$ we have:

Fact 4. $\{s_1, \ldots, s_{k_{j^*}}\} \supseteq \{s_{\sigma(1)}, \ldots, s_{\sigma(i_j)}\}$.

Thus for every $j = 1, \ldots, q$:

$$\begin{aligned} \sum_{i=1}^{i_j} x_{\sigma(i)} &= \sum_{i=1}^{i_j} (\mu_{\sigma(i)} - a_{\sigma(i)})/b_{\sigma(i)} \quad \text{(Fact 2)} \\ &= \sum_{k=1}^{k_{j^*}} c_k(\lambda_k) \quad \text{(Fact 3)} \\ &= \text{the maximum flow value achievable through } \{s_1, \ldots, s_{k_j^*}\} \quad \text{(Fact 1)} \\ &\geqslant \text{the maximum flow value achievable through } \{s_{\sigma(1)}, \ldots, s_{\sigma(i_j)}\} \\ &\quad \text{(Fact 4)}. \end{aligned}$$

Since $\sum_{i=1}^{i_j} x_{\sigma(i)}$ cannot be greater than the maximum flow value achievable through $\{s_{\sigma(1)}, \ldots, s_{\sigma(i_j)}\}$, $\sum_{i=1}^{i_j} x_{\sigma(i)}$ is equal to the maximum flow value achievable through the sinks $\{s_{\sigma(1)}, \ldots, s_{\sigma(i_j)}\}$.

So far we have proved that the maximum flow **f** yields the in-flow vector **x** such that for each $j = 1, \ldots, q$:

Fact 5. $(x_{\sigma(1)}, \ldots, x_{\sigma(i_j)}) = ((\mu_{\sigma(1)} - a_{\sigma(1)})/b_{\sigma(1)}, \ldots, (\mu_{\sigma(i_j)} - a_{\sigma(i_j)})/b_{\sigma(i_j)})$ is the in-flow sub-vector attaining maximum value achievable through the sinks $\{s_{\sigma(1)}, \ldots, s_{\sigma(i_j)}\}$.

Fact 6. $\mu_{\sigma(1)} \leqslant \mu_{\sigma(2)} \leqslant \cdots \leqslant \mu_{\sigma(n)}$ and $\mu_{\sigma(i_1)}, \mu_{\sigma(i_2)}, \ldots, \mu_{\sigma(i_q)}$ is the subsequence of all distinct values sorted in increasing order.

It is shown by induction on $j = 1, \ldots, q$, using (Fact 5) and (Fact 6) that **x** is the solution of (*Lexico'*) with lexicographically maximized vector $(\mu_{\sigma(1)}, \mu_{\sigma(2)}, \ldots, \mu_{\sigma(n)})$. (Alternatively it follows directly from Theorem 9.1 of [16] which is stated in more general terms.)

First, by (Fact 5), $(x_{\sigma(1)}, \ldots, x_{\sigma(i_1)}) = ((\mu_{\sigma(1)} - a_{\sigma(1)})/b_{\sigma(1)}, \ldots, (\mu_{\sigma(i_1)} - a_{\sigma(i_1)})/b_{\sigma(i_1)})$ is the in-flow sub-vector attaining maximum value achievable through the sinks $\{s_{\sigma(1)}, \ldots, s_{\sigma(i_1)}\}$. So the in-flow vector **y** of any maximum flow of $\hat{G}$ satisfies

$$y_{\sigma(1)} \leqslant x_{\sigma(1)}, \ldots, y_{\sigma(i_1)} \leqslant x_{\sigma(i_1)}.$$

Hence,

$$b_{\sigma(1)} y_{\sigma(1)} + a_{\sigma(1)} \leqslant \mu_{\sigma(1)}, \ldots, b_{\sigma(i_1)} y_{\sigma(i_1)} + a_{\sigma(i_1)} \leqslant \mu_{\sigma(i_1)}. \tag{6.4}$$

Assume this holds for all $k \leqslant j-1$, then

$$b_{\sigma(1)} y_{\sigma(1)} + a_{\sigma(1)} \leqslant \mu_{\sigma(1)}, \ldots, b_{\sigma(i_{j-1})} y_{\sigma(i_{j-1})} + a_{\sigma(i_{j-1})} \leqslant \mu_{\sigma(i_{j-1})}. \tag{6.5}$$

Let $\mathbf{x}'$ be an in-flow vector which gives a strictly better solution to (*Lexico'*) than **x**. Then by (6.5) we must have

$$b_{\sigma(1)} x'_{\sigma(1)} + a_{\sigma(1)} = \mu_{\sigma(1)}, \ldots, b_{\sigma(i_{j-1})} x'_{\sigma(i_{j-1})} + a_{\sigma(i_{j-1})} = \mu_{\sigma(i_{j-1})}, \tag{6.6}$$

and from (Fact 6), there must be t with $\sigma(i_{j-1}+1) \leqslant \sigma(t) \leqslant \sigma(i_j)$ such that

$$b_{\sigma(t)} x'_{\sigma(t)} + a_{\sigma(t)} > \mu_{\sigma(t)} = b_{\sigma(t)} x_{\sigma(t)} + a_{\sigma(t)}.$$

Therefore,

$$(x'_{\sigma(t)} - a_{\sigma(t)})/b_{\sigma(t)} > (x_{\sigma(t)} - a_{\sigma(t)})/b_{\sigma(t)}.$$

But then (Fact 5) implies that there must be t' with $\sigma(i_{j-1}+1) \leqslant \sigma(t') \leqslant \sigma(i_j)$ such that

$$(x'_{\sigma(t')} - a_{\sigma(t')})/b_{\sigma(t')} < (x_{\sigma(t')} - a_{\sigma(t')})/b_{\sigma(t')}.$$

Hence:

$$b_{\sigma(t')} x'_{\sigma(t')} + a_{\sigma(t')} < \mu_{\sigma(t')}. \tag{6.7}$$

(6.6) and (6.7) imply that $\mathbf{x}'$ is a worse solution than **x**, which is a contradiction. Thus we conclude that the in-flow vector **x** of **f** of Step 4 yields the optimal value $(\mu_{\sigma(1)}, \mu_{\sigma(2)}, \ldots, \mu_{\sigma(n)})$ to the problem (*Lexico'*). □

The complexity of the main algorithm, Algorithm **Lexico-Finder**, excluding Step 2, is the same as the complexity of the maximum flow problem on $\hat{G}$. This is since Step 4 requires the maximum flow problem on $\hat{G}$, while Step 1 and Step 3 can be executed in $O(n)$ (where n is the number of sources of G.)

In the following subsection, we describe Subroutine **Breakpoint-Finder** that is called for in Step 2, which finds the breakpoints of $\kappa(\lambda)$ in the time of a single application of the preflow algorithm. It follows that the complexity of Algorithm **Lexico-Finder** is the same as that of the preflow algorithm.

6.6. *The breakpoint algorithm*

When $\mathbf{a} = 0$, Step 2 of Algorithm **Lexico-Finder** is executed using the breakpoint algorithm of [12] (discussed in Subsection 6.4) which uses the *parametric maximum flow algorithm* developed in the same paper. While, the parametric maximum flow algorithm works for any type of parametric capacity function once it satisfies the monotonicity assumption, the breakpoint algorithm requires that parametric capacities are also linear functions. When all parametric capacities are linear, the capacity function $\kappa(\lambda)$ is a piecewise concave function with at most $n-2$ breakpoints since at each breakpoint at least one source of $\hat{G}(\lambda)$ shifts from the sink side to source side of a minimum cut as λ increases.

In our case, each parametric capacity $c_k(\lambda)$ is a piecewise convex linear function with single break point (see Fig. 8). So the minimum cut capacity function $\kappa(\lambda)$ is piecewise linear but not concave in general. The function still has nice properties which allow us to use the breakpoint algorithm of [12] as subroutine in order to solve the problem in the same running time.

The set of breakpoints consists of two types of points (which are not necessarily mutually exclusive). The first type of breakpoints are the node-shifting breakpoints (where some nodes of the network shift from the source side to the sink side of a minimum cut as λ increases). Consequently, the number of node-shifting breakpoints is bounded by the number of nodes in G. The second type of breakpoints are derived from the breakpoints of the parametric capacities, $\{a_k \mid k = 1, \ldots, n\}$. Even if we may have the same minimum cut over some range of λ, there can be a change in the slope of $\kappa(\lambda)$ if the parametric capacity of an arc (s, s_k) in the minimum cut begins to increase from

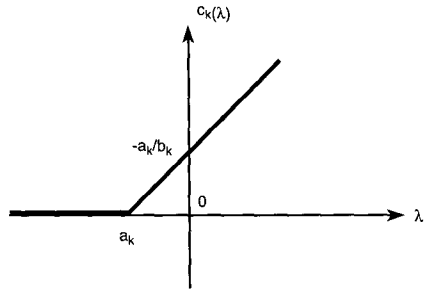

Fig. 8. Capacity function on arc (s, s_k).

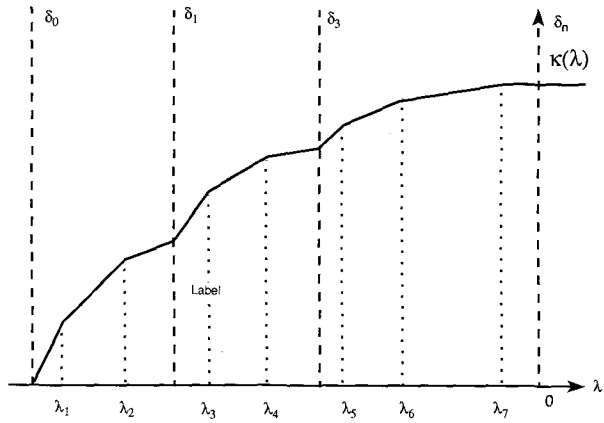

Fig. 9. Minimum cut capcity function $\kappa(\lambda)$ of $\hat{G}(\lambda)$.

zero to positive value at the point $\lambda = a_k$ in the range. Thus the second type breakpoints are from the set $D = \{a_k \mid k = 1, \ldots, n\}$. It may be assumed that $a_k < 0$ for every k since, otherwise, $x_k = 0$ in the optimal solution. Let $\delta_0 \leqslant \delta_1 \leqslant \cdots \leqslant \delta_{n-1}$ be the elements of D sorted in increasing order. In particular $\delta_0 = \min\{a_k \mid k = 1, \ldots, n\}$. To this set we add $\delta_n = 0$. In the proof of Theorem 6.9, μ_k is the derivative of kth objective function at the optimal solution which is nonpositive and $\lambda_k \leqslant \mu_k$ for $k = 1, \ldots, n$. So every breakpoint of $\kappa(\lambda)$ lies on the interval $[\delta_0, \delta_n]$.

An important property of $\kappa(\lambda)$ is that it is concave between two consecutive δ_k's (see Fig. 9). This is because all parametric capacities are linear between two such points. This "piecewise" concavity of $\kappa(\lambda)$ is crucial to get an efficient time bound of Subroutine **Breakpoint-Finder**.

Subroutine Breakpoint-Finder

Step 0: Obtain the sorted sequence $\{\delta_0, \delta_1, \ldots, \delta_{n-1}\}$ of $\{a_1, \ldots, a_n\}$. Let $\delta_n = 0$.

Step 1: Apply the parametric maximum flow algorithm of [12] to $\hat{G}(\lambda)$ with $\lambda = \delta_0, \ldots, \delta_n$ to obtain for each δ_k the maximum flow $\mathbf{f}_k$ and the min-cuts $(W_k, \overline{W}_k)$ such that $|W_k|$ is maximum.
Also apply the parametric maximum flow algorithm to $\hat{G}^R(\lambda)$ to obtain for each δ_k the maximum flow $\mathbf{g}_k$ and the minimum cuts $(\overline{Z}_k, Z_k)$ such that $|\overline{Z}_k|$ is maximum.

Step 2: For each $k = 0, \ldots, n-1$, create $\hat{H}_k(\lambda)$ from $\hat{G}(\lambda)$ by contracting $W_k - \overline{Z}_{k+1}$ and $\overline{Z}_{k+1}$ to single vertices by Procedure **Contraction** $(G(\lambda); W_k, \overline{Z}_{k+1})$ in Subsection 6.4.
Let $\mathbf{f}'_k$ be the flow on $\hat{H}_k(\lambda)$ corresponding to $\mathbf{f}_k$.
Let $\mathbf{g}'_{k+1}$ be the flow on $\hat{H}^R_k(\lambda)$ corresponding to $\mathbf{g}_{k+1}$.

Step 3(a): For each $k = 0, \ldots, n-1$, find the node-shifting breakpoints of $\hat{G}(\lambda)$ on (δ_k, δ_{k+1}) by applying the (main procedure of) breakpoint algorithm of [12] to $\hat{H}_k(\lambda)$ with initial values δ_k, δ_{k+1}, $\mathbf{f}'_k$ and $\mathbf{g}'_{k+1}$. Let L be the set of these breakpoints.

Step 3(b): Let $L' = L \cup \{\delta_0, \delta_1, \ldots, \delta_n\}$. Sort L' in increasing order.

Step 4: Apply the parametric maximum flow algorithm to $\hat{G}(\lambda)$ for all $\lambda \in L'$. Select all the values at which some nodes shift from the sink side to source side and output them as the node-shifting breakpoints of $\hat{G}(\lambda)$.

Theorem 6.10. *Let G have N nodes and M arcs. Subroutine* **Breakpoint-Finder** *finds the node-shifting breakpoints of minimum cut capacity function $\kappa(\lambda)$ of $\hat{G}(\lambda)$ in $O(NM \log(N^2/M))$ steps.*

Proof. The validity of Step 1 through 3 follows from the arguments of Subsection 6.4 which assert that the node-shifting breakpoints of $\hat{G}(\lambda)$ on (δ_k, δ_{k+1}) can be found by applying the breakpoint algorithm of [12] to $\hat{H}_k(\lambda)$ if all parametric capacities are linear on $[\delta_k, \delta_{k+1}]$.

The reason for creating the set L' in Step 3(b), rather than just taking L as the set of all node-shifting breakpoints, is that some of the δ_k's can be node-shifting breakpoints as well. Hence the parametric maximum flow algorithm in Step 4, extract all node-shifting breakpoints.

Let $\hat{N}$ and $\hat{M}$ denote the number of nodes and arcs of $\hat{G}(\lambda)$ respectively. With this notation, $\hat{N} = N + 1$ and $\hat{M} = M + n$. Also let N_k and M_k be the number of nodes and arcs in $\hat{H}_k(\lambda)$ for $k = 0, 1, \ldots, n-1$ respectively.

Since the number of parameter values, δ_k's, is $n \leqslant \hat{N} - 2$, Step 1 also can be done in $O(\hat{N}\hat{M} \log(\hat{N}^2/\hat{M}))$ steps by applying the parametric maximum flow algorithm of [12] to $\hat{G}(\lambda)$ and $\hat{G}^R(\lambda)$.

For a contraction procedure in Step 2, we need to determine the edges in the cut defined by the vertex set to be shrunk into a single vertex. This can be done in $O(\hat{M})$ steps by a breadth first search. The running time of other efforts, deleting loops and combining arcs is also bounded by $O(\hat{M})$ using a standard data structure for network representation. Since n contractions are done and $n \leqslant \hat{N}$, Step 2 requires $O(\hat{M}\hat{N})$ steps.

By Lemma 6.5, Corollary 6.6 and Lemma 6.7, it follows that each node of $\hat{G}(\lambda)$ can participate in at most a single $\hat{H}_k(\lambda)$ as an unshrunk node. Hence $\sum_{k=0}^{n-1} N_k = O(\hat{N})$. The running time of the breakpoint algorithm of [12] for the kth contracted network $\hat{G}_k$ is $O(N_k M_k \log(N_k^2/M_k))$. Let A be the constant coefficient of the running time. Then the total work of Step 3 is:

$$A \sum_{k=0}^{n-1} N_k M_k \log\left(N_k^2/M_k\right) \leqslant A \sum_{k=0}^{n-1} N_k M_k \log\left(2\hat{N}^2/M_k\right)$$
$$\leqslant A \sum_{k=0}^{n-1} N_k \hat{M} \log\left(2\hat{N}^2/\hat{M}\right).$$

The last inequality follows from the fact that the function $f(x) = x \log(K/x)$ is increasing in the range $0 < x \leqslant K/4$ (where K is a positive constant) and $M_k \leqslant \hat{M} \leqslant 2\hat{N}^2/4$. Since $\sum_{k=0}^{n-1} N_k = O(\hat{N})$, Step 3 requires $O(\hat{N}\hat{M} \log(\hat{N}^2/\hat{M}))$ steps.

The size of L' is $O(\hat{N})$; hence, again by the parametric maximum flow algorithm, Step 4 can be done in $O(\hat{N}\hat{M} \log(\hat{N}^2/\hat{M}))$ steps. So the total running time of

Subroutine **Breakpoint-Finder** is $O(\hat{N}\hat{M}\ \log(\hat{N}^2/\hat{M}))$. Since $\hat{N} = N + 1$, $\hat{M} = M + n$ and $n \leqslant M$, the running time is $O(NM\ \log(N^2/(M+n)))$ which is $O(NM\ \log(N^2/M))$. □

Since the running time of Subroutine **Breakpoint-Finder** is $O(NM\ \log(N^2/M))$, and the dominant operations in Algorithm **Lexico-Finder** are the maximum flow on $\hat{G}$ and the calls to Subroutine **Breakpoint-Finder**, it follows that the network resource allocation problem is solvable in $O(NM\ \log(N^2/M))$ steps.

So far the objective function has been assumed to be strictly convex, i.e. $b_k > 0$ for all $k = 1, 2, \ldots, n$. Assume now that the objective function is convex but not strictly convex: hence there is a proper subset $U \subset T$ such that $b_k = 0$ if $s_k \in T - U$.

Theorem 6.3 is still valid since it does not assume the strict convexity of the objective function. So to solve (*Network*) is equivalent to solve (*Lexico'*) which finds a maximum flow of G lexicographically maximizing the sorted sequence of $\{b_k x_k + a_k \mid s_k \in T\}$ in increasing order.

If $s_k \in T - U$, then $b_k x_k + a_k$ is a constant. So it suffices to consider only the elements of $\{b_k x_k + a_k \mid s_k \in U\}$. That is, if $\mathbf{x}$ is an in-flow vector of a maximum flow such that the sorted sequence of $\{b_k x_k + a_k \mid s_k \in U\}$ in increasing order is the lexicographically maximum among all in-flow vectors, then $\mathbf{x}$ is the solution of (*Network*).

This can be done in the following manner:

1. Augment the network G with the arcs (s, s_k) and the parametric capacities $c_k(\lambda)$ for only k's such that $s_k \in U$. Apply Algorithm **Lexico-Finder** to obtain a maximum flow $\mathbf{g}$ and the corresponding in-flow vector $\{y_k \mid s_k \in U\}$.
2. Create the residual network $G_{\mathbf{g}}$ of G with respect to the flow $\mathbf{g}$. Augment $G_{\mathbf{g}}$ with the arcs (s, s_k) for $s_k \in T - U$ and find a maximum flow of the network. Let the corresponding in-flow vector be $\{z_k \mid s_k \in T - U\}$.
3. Output the in-flow vector $\{y_k \mid s_k \in U\} \cup \{z_k \mid s_k \in T - U\}$ as an optimal solution of (*Network*).

The additional work for the general convex case is to find a maximum flow in the residual graph $G_{\mathbf{g}}$. Hence the total running time is the same as that of the strictly convex case.

7. Concluding remarks

We presented new strongly polynomial algorithms for some special cases of discrete and continuous convex separable quadratic optimizations over submodular constraints. It seems that further improvement on the complexities of the algorithms developed here is going to be challenging.

As mentioned earlier, every problem considered in this paper is a special case of the minimum quadratic cost network flow problem; costs are associated only to the flows on the arcs emanating into the sink of the network. This observation naturally leads us to the important open question of the strong polynomiality of (general) minimum quadratic cost network flow problem. This problem seems to be very challenging. Even if the number of arcs with quadratic cost is fixed, the question still remains open.

References

[1] R. Baldick and F.F. Wu, "Efficient integer optimization algorithms for optimal coordination of capacitators and regulators," *IEEE Transactions on Power Systems* 5 (1990) 805–812.

[2] M.J. Best and R.Y. Tan, "An $O(n^2 \log n)$ strongly polynomial algorithm for quadratic program with two equations and lower and upper bounds," Research Report CORR 90-04, Department of Combinatorics and Optimization, University of Waterloo (1990).

[3] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest and R.E. Tarjan, "Time bounds for selection," *Journal of Computer and Systems Sciences* 7 (1972) 448–461.

[4] P. Brucker, "An $O(n)$ algorithm for quadratic knapsack problems," *Operations Research Letters* 3 (1984) 163–166.

[5] W. Cook, A.M.H. Gerards, A. Schrijver and E. Tardos, "Sensitivity results in integer linear programming," *Mathematical Programming* 34 (1986) 251–264.

[6] S. Cosares and D.S. Hochbaum, "Strongly polynomial algorithms for the quadratic transportation problem with fixed number of sources," *Mathematics of Operations Research* 19(1) (1994) 94–111.

[7] M.E. Dyer and A.M. Frieze, "On an optimization problem with nested constraints," *Discrete Applied Mathematics* 26 (1990) 159–173.

[8] G.N. Frederickson and D.B. Johnson, "The complexity of selection and ranking in $X + Y$ and matrices with sorted columns," *Journal of Computer and Systems Sciences* 24 (1982) 197–208.

[9] S. Fujishige, "Lexicographically optimal base of a polymatroid with respect to a weight vector," *Mathematics of Operations Research* 5 (1980) 186–196.

[10] S. Fujishige, *Submodular Functions and Optimization*, Annals of Discrete Mathematics 47 (North-Holland, Amsterdam, 1991).

[11] H.N. Gabow and R.E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Journal of Computer and System Sciences* 30 (1985) 209–221.

[12] G. Gallo, M.E. Grigoriadis and R.E. Tarjan, "A fast parametric maximum flow algorithm and applications," *SIAM Journal of Computing* 18 (1989) 30–55.

[13] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," in: *Proceedings of the 18th Annual ACM Symposium on Theory of Computing* (1986) pp. 136–146.

[14] F. Granot and J. Skorin-Kapov, "Some proximity and sensitivity results in quadratic integer programming," Working Paper No. 1207, University of British Columbia (1986).

[15] F. Granot and J. Skorin-Kapov, "Strongly polynomial solvability of a nonseparable quadratic integer program with applications to toxic waste disposal," Manuscript (1990).

[16] H. Groenevelt, "Two algorithms for maximizing a separable concave function over a polymatroidal feasible region," Technical Report, The Graduate School of Management, University of Rochester (1985).

[17] D.S. Hochbaum, "On the impossibility of strongly polynomial algorithms for the allocation problem and its extensions," in: *Proceedings of the 1st Integer Programming and Combinatorial Optimization Conference* (1990) pp. 261–274; D.S. Hochbaum, "Lower and upper bounds for the allocation problem and other nonlinear optimization problems," *Mathematics of Operations Research* 19 (1994) 390–409.

[18] D.S. Hochbaum and J.G. Shanthikumar, "Convex separable optimization is not much harder than linear optimization," *Journal of ACM* 37 (1990) 843–862.

[19] D.S. Hochbaum, R. Shamir and J.G. Shanthikumar, "A polynomial algorithm for an integer quadratic nonseparable transportation problem," *Mathematical Programming* 55(3) (1992) 359–372.

[20] T. Ibaraki and N. Katoh, *Resource Allocation Problems: Algorithmic Approaches* (MIT Press, Cambridge, MA, 1988).

[21] N. Megiddo and A. Tamir, "Linear time algorithms for some separable quadratic programming problems," *Operations Research Letters* 13 (1993) 203–211.

[22] G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization* (Wiley, New York, 1988).

[23] A. Tamir, "A strongly polynomial algorithm for minimum convex separable quadratic cost flow problems on series-parallel networks," *Mathematical Programming* 59 (1993) 117–132.

[24] E. Tardos, "A strongly polynomial algorithm to solve combinatorial linear programs," *Operations Research* 34 (1986) 250–256.