Solving Linear Cost Dynamic Lot Sizing Problems in O(n log n) Time

Ravindra K. Ahuja[1] and Dorit S. Hochbaum[2]

Abstract

In this paper, we study capacitated dynamic lot sizing problems with or without backorders, under the assumption that production costs are linear, that is, there are no set up costs. These two dynamic lot sizing problems (with or without backorders) are minimum cost flow problems on an underlying network that possess a special structure. We show how the well-known successive shortest path algorithm for the minimum cost flow problem can be used to solve these problems in $O(n^2)$ time, where n is the number of periods in the dynamic lot sizing problems, and how with the use of dynamic trees we can solve these problems in O(n log n) time. Our algorithm also extends to the dynamic lot sizing problem with integer variables and convex production costs with running time O(n log n log U), where U is the largest demand value.

1. Introduction

In this paper, we study the lot sizing problem (Wagner and Whitin [1958]) which has time-varying demands, costs and capacities but there are no set up costs. In today's manufacturing environment, the objective is to reduce and avoid set up costs whenever possible. The strategy of using "outsourcing" or "subcontracting" has a number of advantages such as shortening the supply chain and mitigating the effect of uncertainties. One major feature of outsourcing is that manufacturing on that basis is based primarily on variable cost and thus the cost of production has no set up costs. With linear (or convex) production costs, the lot sizing problem - even in the presence of capacities - can be represented as a network flow problem, and thus can be solved efficiently in polynomial time. Our purpose here is to show that this problem can be solved extra efficiently in O(n log n) time that is dramatically better than that obtained by an algorithm that solves the problem as a pure network flow problem. On the contrary, the capacitated lot sizing problem with fixed charges is NP-hard (Florian et al. [1980], and Bitran and Yanasse [1982]). Our algorithm has also been used recently to improve the performance of heuristics for multi-item capacitated lot sizing problem with fixed charges by Federgruen et al. [2005].

The lot sizing problem considered here has a prescribed demand d_i of a single item for each of the n periods i = 1, 2, … , n which must be met by either producing x_{0i} items in period i and/or by drawing

[1] Dept. of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611.
[2] Dept. of IE and OR, and Haas School of Management, University of California, Berkeley, CA 94720.

upon the inventory $x_{i-1,i}$ from the previous period. Figure 1(a) models the situation without backordering and Figure 1(b) models it for the backordering case. In the network, each node i denotes the period i and its demand is d_i. Node 0 denotes the *source* node and each arc (0, i) emanating from this node is called a *production arc* and the flow on this arc represents the production quantity in period i. The supply of the source node equals $\sum_{i \in N} d_i$. Each arc (i-1, i) for $2 \leq i \leq n$ is an inventory-carrying arc and the flow on this arc represents the inventory carried from period i-1 to period i. The flows on arcs (i, i-1) for i = 2, 3, ... , n represent backordering amounts. Each arc (i, j) in the network has two associated values: c_{ij}, the unit cost of flow on arc (i, j), and u_{ij}, the capacity of the arc (i, j). The cost $c_{i,i+1}$ is the cost of carrying one unit of inventory from period i to period i+1. The cost $c_{i,i-1}$ is the cost of backordering one unit from period i to period i-1. Further, the cost $c_{0,i}$ is the cost of production of one unit in period i. The respective values of $u_{i,i+1}$, $u_{i,i-1}$ and $u_{0,i}$ are the inventory-carrying capacity, the bound on backordering capacity and the production capacity. For simplicity, we will assume that all the costs (production, inventory-carrying, and backordering) are nonnegative and that without loss of generality the total production capacity is at least as large as the total demand, as otherwise there is no feasible solution.

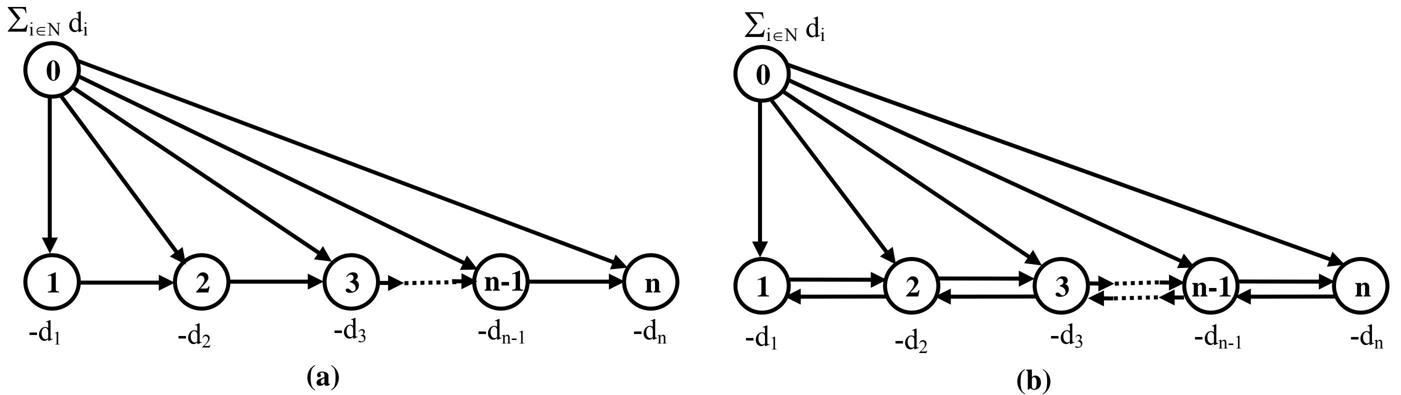

Figure 1. Flow networks for lot sizing problems.
(a) Lot sizing without backordering. (b) Lot sizing with backordering.

The network flow models shown in Figure 1 are special cases of the minimum cost flow problem (Ahuja et al. [1993]). The number of arcs in the network shown in Figure 1(a) is 2n-1 and in the network shown in Figure 1(b) is 3n-2, both are considerably fewer than the maximum number of arcs possible, which is n^2. These networks are called *sparse*. For sparse minimum cost flow problems with n nodes and

m arcs, the best strongly polynomial algorithm currently known, due to Orlin [1988], takes O(m log n (m + n log n)) time. Since for both the lot sizing problems, m = O(n), this algorithm solves them in $O(n^2 \log^2 n)$ time. Our main contribution in this paper is to show that the minimum cost flow problems in Figure 1 can be solved in O(n log n) time which obtains a speedup of a factor of O(n log n) over the best currently available algorithm.

A special case of the problem studied by us in this paper has been recently considered by Sedeño-Noda et al. [2004]. This paper considers the lot sizing problem with linear production costs problem where the inventory carrying capacity is bounded and suggests an O(n log n) algorithm. Our paper generalizes this result on several fronts: we allow backordering to satisfy demands, and allow upper bounds on production capacities as well as backordering quantities, and solves the generalized problem without worsening the running time.

2. Dynamic Lot sizing without Backorders in $O(n^2)$

The successive shortest path algorithm for minimum cost network flow maintains a solution x that satisfies the non-negativity and capacity constraints but may violate the mass balance constraints of the nodes; such a solution x is known as a *pseudoflow* (Ahuja et al. [1993]). For any pseudoflow x, the *imbalance* of node $i \in N$ is defined as $e(i) = b(i) + \Sigma_{\{j:(j,i)\in A\}} x_{ji} - \Sigma_{\{j:(i,j)\in A\}} x_{ij}$, where b(i) is the supply of node i if b(i) is positive and demand of node i otherwise. If e(i) > 0 for some node i, we refer to e(i) as the *excess* of node i; and if e(i) < 0, we call –e(i) the node's *deficit*. The algorithm also maintains a residual network G(x) corresponding to a pseudoflow x, where each arc $(i, j) \in A$ is replaced by two arcs (i, j) and (j, i). The arc (i, j) has *cost* c_{ij} and *residual capacity* $r_{ij} = u_{ij} – x_{ij}$, and the arc (j, i) has cost $c_{ji} = -c_{ij}$ and residual capacity $r_{ji} = x_{ij}$. We refer to an arc $(i, j) \in A$ as a *regular arc* and its reversal (j, i) as a *reverse arc*. The residual network consists only of arcs with positive residual capacities.

```
algorithm successive-shortest-path;
begin
        set x: = 0;
        while x is not a flow do
        begin
                select an excess node s and a deficit node t;
                identify a shortest path P from node s to node t in G(x);
                augment maximum possible flow along P and update G(x);
        end;
end;
```

We refer to the network in Figure 1(a) as G = (N, A). We start with x = 0 for which node 0 is the only excess node and all other nodes are deficit nodes. Our algorithm runs n iterations as follows. At iteration i, nodes 1, …, i-1 have their demands satisfied and we send flow along shortest paths from node 0 to node i until the demand of node i is met. To identify shortest paths to deficit nodes quickly, we maintain a set of all directed paths to deficit nodes arranged in the increasing order of paths costs. Each directed path in G from the source node 0 to any node i in G is of the form 0-k-(k+1)-(k+2)- ... -i; we refer to this path as P_{ki}. Let ***P***(i) denote the set of *all* directed paths in the residual network G(x) from node 0 to node i with respect to the flow x. For example, the set $\{P_{14}, P_{24}, P_{34}, P_{44}\}$ gives all directed paths in G from node 0 to node 4 in Figure 1(a). For any path P_{ki}, let $c(P_{ki})$ denote the cost of the path, and $r(P_{ki}) := \min\{u_{kl} - x_{kl} : (k, l) \in P_{ui}\}$ be the residual capacity of the path. For every $P_{ki} \in$ ***P***(i) we maintain the following invariants: (i) $r(P_{ki}) > 0$, and (ii) paths in ***P***(i) are arranged in the non-decreasing order of the path costs. Our implementation of the successive shortest path algorithm is given below.

```
algorithm dynamic lot sizing;
begin
    set x := 0;
    compute node imbalances e(i)'s;
    for i := 1 to n do
    begin
        compute P(i);
        while e(i) > 0 do
        begin
            select the lowest cost path P_ui ∈ P(i);
            compute r(P_ui) : = min{u_kl - x_kl : (k, l) ∈ P_ui};
            augment δ = min{r(P_ui), e(i)} units of flow along P_ui;
            update P(i), the flow x and node imbalances e(i);
        end;
    end;
end;
```

The correctness of the algorithm follows from the correctness of the successive shortest path algorithm (see Ahuja et al. [1993]). To analyze the complexity of the algorithm, we describe how to compute $\boldsymbol{P}(i)$ from $\boldsymbol{P}(i-1)$ for any node i. Clearly, $\boldsymbol{P}(1) = \{P_{11}\}$. Since every directed path in the residual network from node 0 to i will be one of the following paths: P_{1i}, P_{2i}, P_{3i}, … , $P_{i-1,i,}$ and P_{ii}, $\boldsymbol{P}(i)$ is determined by appending the arc (i-1, i) to each path $\boldsymbol{P}(i-1)$ and adding the path P_{ii}. Since paths in $\boldsymbol{P}(i-1)$ are already sorted in the non-decreasing order of the path costs, appending the arc (i-1, i) to each path preserves the sorted order. Adding the new path P_{ii} to $\boldsymbol{P}(i)$ is done by inserting the path P_{ii} in its proper sorted position in $\boldsymbol{P}(i)$. Clearly, these operations take O(n) time.

In the **while** loop of the algorithm, selecting the lowest cost path P_{ui} in $\boldsymbol{P}(i)$ takes O(1) time, computing the residual capacity of any path P_{ui}, $r(P_{ui})$, takes O(n) time, and augmenting $\delta = \min\{r(P_{ui}), e(i)\}$ units of flow along the path takes O(n) time. Augmentation may reduce the residual capacities of several paths in $\boldsymbol{P}(i)$ to zero and these paths must be removed from $\boldsymbol{P}(i)$. If the residual capacity of the arc $(l, l+1)$ becomes zero then the residual capacity of each path P_{pi} with $p \leq l$ becomes zero, and we must remove it from $\boldsymbol{P}(i)$. As $\boldsymbol{P}(i)$ contains at most n-1 paths, this step takes O(n) time. Thus, each execution of the **while** loop takes O(n) time. Since each execution of the **while** loop either reduces the imbalance of node i to zero (which can happen at most n times) or reduces the residual capacity of an arc to zero (which can happen at most (2n-1) times as the network has (2n-1) arcs) and flow is never decreased; thus,

the number of executions of the **while** loop are bounded by (2n-1), and the total time taken by the algorithm is $O(n^2)$. We summarize the preceding discussion as the following theorem:

Theorem 1. *The successive shortest path algorithm solves the lot sizing problem without backorders in $O(n^2)$ time.*

4. Solving the Dynamic Lot sizing with Backordering in $O(n^2)$

We refer to the network shown in Figure 1(b) as $G' = (N', A')$ noting that $|A'| = 3n-2$. Our algorithm for the backordering case is the same as described in the previous section except that the sets $\boldsymbol{P}(i)$ of all the directed paths in the residual network $G'(x)$ from node 0 to node i now include "backward paths". With respect to the zero flow x, any directed path in $G'(x)$ from node 0 to node i is either of the form of a *forward path* P_{ki} , 0-k-(k+1)-(k+2)-...-i for some $k \le i$, or it is a *backward path* P_{li} of the form 0-l-(l-1)-(l-2)-...-i for some $l > i$. We store $\boldsymbol{P}(i)$ as the union of the set $\overline{\boldsymbol{P}}_i$ of forward paths from source to node i, and the set $\underline{\boldsymbol{P}}_i$ of backward paths from source to node i. For example, in G′, $\boldsymbol{P}(1) = \{P_{11}, P_{21}, P_{31}, \dots , P_{n1}\}$ where $\overline{\boldsymbol{P}}_1 = \{P_{11}\}$ and $\underline{\boldsymbol{P}}_1 = \{P_{21}, P_{31}, \dots , P_{n1}\}$. Both $\overline{\boldsymbol{P}}_i$ and $\underline{\boldsymbol{P}}_i$ are maintained as ordered sets sorted in the ascending order of path costs. Thus, to identify a shortest path in $\boldsymbol{P}(i)$, we select the lower cost between the first path in $\overline{\boldsymbol{P}}_i$ and the first path in $\underline{\boldsymbol{P}}_i$.

When we increment i-1 to i, then both $\overline{\boldsymbol{P}}_i$ and $\underline{\boldsymbol{P}}_i$ are recomputed. As before, $\overline{\boldsymbol{P}}_i$ is obtained by appending the arc (i-1, i) to each path $P_{k,i-1} \in \overline{\boldsymbol{P}}_{i-1}$ and adding the path P_{ii}. Next observe that the residual network $G'(x)$ may contain two arcs from node i-1 to node i: a "regular" inventory-carrying arc from node i-1 to node i, and a "reverse" arc which is the reversal of the backordering arc from node i-1 to node i with positive flow. Since our algorithm augments flow along shortest paths, we use the arc with the smaller cost to append to the paths in $\overline{\boldsymbol{P}}_{i-1}$. Since a regular arc has a nonnegative cost, its reverse arc, if exists, is always selected as it has a nonpositive cost. Since we append the same arc to each path in $\overline{\boldsymbol{P}}_{i-1}$, the extended paths in $\overline{\boldsymbol{P}}_i$ remain in the sorted order. In addition, we add the path P_{ii} to the set $\overline{\boldsymbol{P}}_i$. Thus, we can compute the set $\overline{\boldsymbol{P}}_i$ in O(n) time. To recompute $\underline{\boldsymbol{P}}_i$ from $\underline{\boldsymbol{P}}_{i-1}$, we consider each path $P_{l,i-1} \in \underline{\boldsymbol{P}}_{i-1}$,

and delete it if l = i; otherwise, we replace $P_{l,i-1}$ by P_{li}. All the steps needed to recompute $\underline{P}_i$ from $\underline{P}_{i-1}$ take a total of O(n) time.

We next describe how to update these sets as flow is augmented along a shortest path, say P_{ui}. If the flow δ augmented along the path P_{ui} satisfies $\delta < r(P_{ui})$, then the flow augmentation keeps the sets $\overline{P}_i$ and $\underline{P}_i$ intact. If, however, $\delta = r(P_{ui})$, then after the flow augmentation the residual capacities of some arcs reduce to zero. In that case, the residual capacities of some paths in $\overline{P}_i$ and $\underline{P}_i$ reduce to zero and these paths need to be removed. These are four cases to consider:

Case 1: *P_{ui} is a forward path and (p, q) is a regular arc.* Recall that we use a regular arc in extending paths when there is no reverse arc in the residual network. Hence, when the arc (p, q) becomes saturated, the residual capacity of each path in $\overline{P}_i$ containing the arc (p, q) become zero. Therefore, each path $P_{ki} \in \overline{P}_i$ with $k \leq p$ must be deleted which takes O(n) time.

Case 2: *P_{ui} is a forward path and (p, q) is a reverse arc.* We replace the reverse arc (p, q) by the regular arc (p, q). Then, $\alpha = c_{p,p+1} + c_{p+1,\, p}$, is the difference in the costs of two arcs. This change would increase the cost of every path in $\overline{P}_i$ containing arc (p, q) by α. To update the set $\overline{P}_i$, we first partition the ordered set $\overline{P}_i$ into two ordered set $\overline{P}_i^1$ and $\overline{P}_i^2$, the first subset includes all paths in $\overline{P}_i$ that contain the arc (p, q), and the second subset contains the remaining paths. We then increase the cost of each path in $\overline{P}_i^1$ by α which preserves its sorted order. The two sorted sets $\overline{P}_i^1$ and $\overline{P}_i^2$ are then merged into a single sorted set to obtain the updated set $\overline{P}_i$. Both the steps described above can be performed in O(n) time.

Case 3: *P_{ui} is a backward path and (p, q) is a regular arc.* In this case, saturating arc (p, q) disconnects all backward paths in $\underline{P}_i$ containing (p, q) and they are deleted from $\underline{P}_i$. This update is done, as in Case 1, in O(n) time. When arc (p, q) is saturated, the production in periods p, p+1, … , n, cannot be used to meet the demands in period $i \leq q < p$, yet it does not preclude meeting the demand of period p by the productions in periods p, p+1, … , n. We thus define a set of such paths, LIST[p], that contains an ordered list of paths to node p which are subpaths of $\underline{P}_{i-1}$ from node n to node p. At a later iteration, when we

examine node p to meet its demand, the set of paths in LIST[p] are also considered for augmentation. Creating this list and deleting the paths can be done in O(n) time.

Case 4: *P_{ui} is a backward path and (p, q) is a reverse arc.* It is easy to see that this case cannot occur since the reverse arc (p, q) can be created only when we send positive flow on the inventory-carrying arc (q, p). The order in which we meet the demands of nodes 1, 2, 3, …., n, ensures that when we augment flow to node i, flow on any inventory-carrying arcs, (i, i+1), (i+1, i+2), …. , (n-1, n) is zero. Hence this case will not occur during the execution of the algorithm.

Therefore, each execution of the **while** loop takes O(n) time and either reduces the deficit of a node to zero or reduces the residual capacity of an arc to zero. Once a residual capacity of an arc becomes zero, it remains zero. Hence, the number of executions of the **while** loop is O(n). This shows that the algorithm runs in $O(n^2)$ time, establishing the following theorem:

Theorem 2. *The successive shortest path algorithm solves the lot sizing problem with backorders in $O(n^2)$ time.*

4. The O(n log n) Implementations of the Lot Sizing Algorithms

We first consider the lot sizing problem where inventory carrying arcs have unbounded arc capacities, there are no backorders and the solution needs to honor the production capacities, and show that this uncapacitated problem can be solved using red-black trees in O(n log n) time. We then address the capacitated version that includes the backordering case where both inventory carrying arcs and backordering arcs may have finite capacities and show that with the use of dynamic trees data structure we can again solve the problem in O(n log n) time.

4.1 The Uncapacitated Lot Sizing With Forward Production

The red-black tree data structure (discussed in detail by Cormen et al. [2001]) defined on a set of object S allows each of the following operations to be performed in O(log n) time:

insert (i): add an object i to the tree S;
delete (i): delete an object i from the tree S;
findmin(S): output an object i_{min} of minimum key value in the tree S; that is, $key[i_{min}] \leq key[i]$ for all $i \in S$.

At iteration i, the red-black tree contains as keys the sorted paths costs $C(P_{ji})$ for the subset of periods with positive production capacity in {1, 2, …, i-1}. We call this tree the *forward tree*. We claim that the sorting of keys in the forward tree is a suborder of the sorting for j = 1, 2, …, n of the key values $C_j = C(P_{jn}) = c_{0j} + c_{j,j+1} + \ldots + c_{n-1,n}$ - namely, the cost of producing a unit in period j and carrying it in inventory until period n. To see this, note that $C(P_{q,i}) = C_q - \sum_{j=i+1}^{n} c_{j-1,j}$, where the sums $\sum_{j=i+1}^{n} c_{j-1,j}$ are constants which are computed a-priori in time O(n). Similarly, we claim that the sorting of the analogous backward tree, used in the next section, is similarly a suborder of the sorting of the key values $C'_j = C(P_{j1}) = c_{0j} + c_{j,j-1} + \ldots + c_{2,1}$ - namely, the costs of producing a unit at period *j* and backordering the unit to period 1. Again, this is proved by observing that the cost of the path $C(P_{q,i})$ is $C'_q - \sum_{j=2}^{i} c_{j,j-1}$ where the sums $\sum_{j=2}^{i} c_{j,j-1}$ are constants to be computed a-priori in time O(n). Here is how the a-priori computing is done. We first introduce the notation:

$C[i+1, n] = \sum_{j=i+1}^{n} c_{j-1,j}$, and

$C'[2, i] = \sum_{j=2}^{i} c_{j,j-1}$.

The recursive procedure computing the sums is then

$C[n-1, n] = 0$ and $C[i, n] = C[i+1, n] + c_{i,i+1}$ for i = 1, …, n-2, and,

$C'[2, 2] = c_{2,1}$, and $C'[2, i] = C'[2, i-1] + c_{i,i-1}$ for i = 3,…, n.

Recall that $u^r_{ij} = u_{ij} - x_{ij}$ is the residual capacity on arc (i, j). The procedure works with arrays FLOW(j) for j = 1, ..., n that are initialized as empty, and at termination contain a collection of flow paths for flows produced at j and going to future periods, each with associated flow amount.

```
procedure Uncapacitated Lot Sizing;
begin
    F := Ø is a forward red-black tree with key[i] = C_i for all i ∈ F;
    i := 1;
    until i = n do
    begin
        insert(i) in forward tree F with key[i] = C_i;
        q := findmin(F);
        δ := min{d_i, u^r_0q}, d_i := d_i - δ, and u^r_0q := u^r_0q - δ;
        store δ_ij = δ in FLOW(q) array.
        if u^r_0q = 0 then delete q from its respective tree;
        if d_i > 0 then repeat, else i := i+1 and repeat;
    end;
end;
```

The operation of *findmin*(F) is applied at most 2n times as each time it is applied either demand of a node becomes zero or the production capacity of a node is saturated. The *insert* and *delete* operations can be applied up to n times each at complexity of up to O(log n) per operation. Thus the total complexity of *Uncapacitated Lot Sizing* is O(n log n).

When this procedure terminates, the arrays FLOW() contain the information, for each production period, how much to produce for each (future) period. We show next how to recover the flows on each arc from the arrays FLOW(j) for j = 1, ..., n in linear time. The total number of entries in these arrays is no more than 2n, because whenever a value is added to the array, either the demand or the production capacity of one node is exhausted. We now create an auxiliary graph on the nodes 1, …, n and up to 2n arcs representing FLOW() . For each node i and $j \geq i$, we place an arc (i, j) that corresponds to the flow path in FLOW(i) with flow δ_{ij}. (For j = i, we place a self loop adjacent to node i with flow value δ_{ii}.)

Starting from j = 1, once the flow on (j–1, j) has been determined we evaluate the flow on (j, j+1) noting that the flow on (j, j+1) is equal to the flow on (j–1, j) minus the flow on the arcs terminating at j plus the flow on arcs beginning at j. As illustrated in Figure 2, δ_{ik} contributes both to the flows on (j–1, j) and on (j, j+1), δ_{pj} contributes to the flow on (j–1, j) but not on (j, j+1), δ_{jq} contributes to the flow on (j, j+1) but not on (j–1, j). This leads to the following linear time procedure.

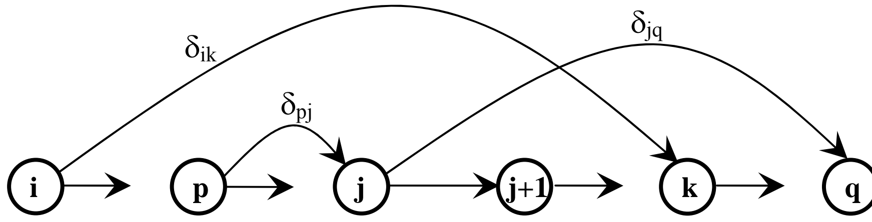

Figure 2. The flow recovery auxiliary graph.

procedure *Flow-Recovery-Forward*;
begin
$x_{0,1}$:= 0; {so that $x_{j-1,j}$ is well defined for j = 1}
for j := 1 **to** n **do**
begin
$x_{j,j} = \delta_{j,j}$;
$x_{j,j+1} = x_{j-1,j} - \sum_{q<j} \delta_{qj} + \sum_{q>j} \delta_{jq}$;
end;
end;

Since the number of terms in the sums for each j is the number of paths in FLOW(j) and no term is repeated in more than one iteration, the entire procedure takes O(n) time.

4.2 The Capacitated Lot Sizing With Backordering

The backordering case creates reverse residual capacities with finite capacities, and thus must be addressed by an algorithm that can solve efficiently the capacitated case. For this purpose, we introduce the use of dynamic trees to track the residual flows. Red-black trees can still be used as before to maintain the array of paths costs, but for simplicity sake, we will use only dynamic trees with **two** sets of keys, maintaining the paths costs on one set of keys and the residual flows on the second set.

Dynamic trees is a data structure used to represent trees. Trees representing paths [a, b] are called *solid trees*. In our procedure, each dynamic tree is used to represent a path or a contiguous subinterval in [1, ..., n] and is therefore a solid. The nodes on the path are represented in a so-called *symmetric order* with the endpoints of the paths being the leftmost leaf and rightmost leaf, respectively. For additional properties and construction of dynamic trees, we refer the reader to Tarjan [1983].

One can use dynamic trees to find a minimum key value along a subpath; add a constant to all keys along a subpath; cut off a subpath of the form [a, p] or [p, b] (can cut off any subpath, but we will not use

this operation); and perform the other operations specified for red-black trees. Each of these operations can be executed in amortized time O(log n). Formally, in addition to the operations of *insert* and *delete* supported also by red-black tree, the following operations are supported by a dynamic tree:

split[a, p]: Split the tree into two trees, one corresponding to the subpath [a, p] and the other to the complement subpath [p+1, b]. (Following the split both these trees are dynamic trees.)
addvalue[a, p] (C): Add constant C to all nodes on the path [a, p].
findmin[k, l]: Find an object i_{min} of minimum key value on the subpath [k, *l*] in the tree.

We maintain throughout the procedure two types of dynamic trees, F and B, for handling the forward and backward paths information. Each of the dynamic trees has two sets of keys with $key^1[j] = C_j$ for the forward tree representing the respective path costs (as for F in the previous section) which is a suborder of the sorting of C_j's. Similarly, the backward tree B has $key^1[j] = C'_j$ representing path costs which form a suborder of the sorting of C'_js. We recover the actual path costs as discussed in the previous section by subtracting a constant. The second sets of key values in both trees maintain the residual capacities. The key values are initialized as $key^2[i] = u^r_{i-1,i}$ for i in F and $key^2[i] = u^r_{i,i-1}$ for i in B. Initially, these are set equal to the capacity upper bounds, which if infinite, are set to the value $M = \sum_{i=1}^{n} d_i$.

At iteration i, F contains the set of objects {a, …, i} (or path [a, i] for a ≥ 1). The backward tree(s) may however be partitioned to several collections of paths of the form {i+1, …, b} (or path [i+1, b] for b ≤ n), each of which is a backward tree. Only one of these, denoted by B, contains the set of objects which is the "active" backward tree. The other collections are considered non-active when there is at least one backward arc (b+1,b) that is saturated. These may become active and originate backorder flows when i ≥ b+1. These non-active trees are maintained in an array, called *list*, which corresponds to a partition of the path [i+1, n] to subintervals containing a list of backward dynamic trees stored and sorted according to the left endpoint of their respective paths. Thus, at iteration i, *list* = (a_1, a_2, …, a_l) corresponds to the partition, ([i+1, a_1-1], [a_1, a_2-1], [a_2, a_3-1],……, [a_l, n]), where the backward arcs (a_i, a_i-1) are saturated. This list is similar to the set LIST[] in Case 3 of Section 3. Initially, *list* = (). In Figure 3, the crossed backward arcs represent the backward arcs that are saturated.

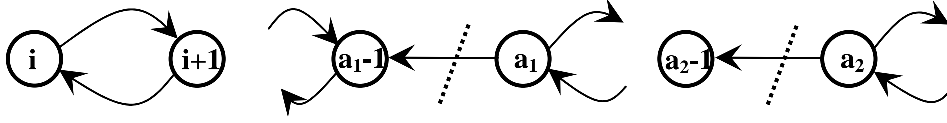

Figure 3. The use of list() at iteration i.

The operations that can be applied to either set of keys are denoted by $findmin^k$ and $addvalue^k$ referring to operations applied to the set of keys k = 1, 2. A boolean parameter, BACK(j), associated with arc (j-1, j) is 1 (true) if there is positive backward flow on this arc, namely $x_{j,j-1} > 0$, and is 0 (false) otherwise. The cost of sending forward flow on such arc is $-c_{j,j-1}$ if BACK(j) = 1 and it is $c_{j-1,j}$ otherwise.

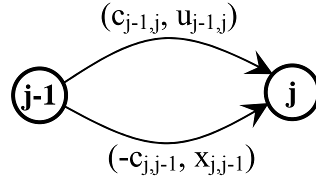

Figure 4. Illustrating the BACK array when BACK(j) = 1.

Figure 4 shows that there are two forward residual arcs when BACK(j) =1, one of negative cost and the other of positive cost. The first has capacity $x_{j,j-1}$ the other has capacity $u_{j-1,j}$. The cost differential between the forward arc and the residual backward arc is $-c_{i,i-1} - c_{i-1,i.}$

4.3 Complexity Analysis

For each node i, we apply *delete* and *insert* once. The operations of $findmin^1$ and $findmin^2$ are applied once for each iteration i in which either the demand of node i is equal to 0, or the residual supply u^r_{0q} of node q is equal to 0, or a bottleneck arc has been encountered. The first two can happen throughout the procedure at most n times each. A reverse arc can become a bottleneck once, and a forward arc can become bottleneck at most twice (once for the reversing of the backflow and once for the capacity of the forward flow). So each of the O(n) arcs contribute at most twice to these operations. So all operations are applied at most O(n) times throughout the procedure for a total complexity of O(n log n).

procedure *Capacitated-Lot-Sizing*;
begin

initialize F := Ø and B := {1, 2, …, n} a backward dynamic tree with $key^1[i] = C'_i$ and $key^2[i] = u^r_{i,i-1} = u_{i,i-1}$ for all $i \in B$;

i := 1

until i = n **do**

begin

delete(i) from backward tree B with $u^r_{i,i-1} = key^2[i]$ and *insert(i)* in forward tree F with $key^1[i] = C_i$ and $key^2[i] := u_{i-1,i}$;

if $x_{i,i-1} = u_{i,i-1} - u^r_{i-1,i} > 0$ **then** BACK(i) = 1 **else** BACK(i) = 0;

if BACK(i) = 1 **then** $addvalue^1[a, i](-c_{i,i-1} - c_{i-1,i.})$ and $key^2[i] := x_{i,i-1}$;

$i_f := findmin^1(F)$ and $i_b = findmin^1(B)$;

if $key^1[i_f] - \sum_{j=i+1}^{n} c_{j-1,j} \le key^1[i_b] - \sum_{j=2}^{i} c_{j,j-1}$ **then** q := i_f and go to *Update* F **else**

q := i_b and go to *Update* B;

if $u^r_{0,q} = 0$ **then** *delete*(q) from its respective tree;

if $d_i > 0$ **then** repeat **else**

if B is empty (of the form [i, i]) **then** let B be the leftmost subpath in the partition *list* [i + 1, a_1 -1]; i := i + 1 and repeat;

end;

end;

procedure *Update* F;
begin

p := $findmin^2[q, i]$ {the bottleneck capacity $r(P_{qi})$} and $\Delta_F := key^2[p] = u^r_{p,p+1}$;

$\delta := \min\{d_i, u^r_{0q}, \Delta_F\}$;

$d_i := d_i - \delta$, $u^r_{0q} := u^r_{0q} - \delta$, $addvalue^2[q, i]\ (-\delta)$;

if $\delta = \Delta_F = u_{p,p+1}$ **then**

if BACK(p + 1) = 0 **then** *split*[a, p] from F {(p, p+1) is saturated}

else set BACK(i) = 0 and $addvalue^1[a, p](c_{p,p+1} + c_{p+1,p})$, and $key^2[p] := u_{p,p+1}$;

{So *F* is [p + 1, i] and a := p+1}

end;

procedure *Update B*;
begin

$p' := findmin^2[i, q]$ {note that q > i};

$\Delta_B := key^2[p'] = u^r_{p',p'-1}$; {$\Delta_B$ is the bottleneck capacity $r(P_{qi})$}

$\delta := \min\{d_i, u^r_{0q}, \Delta_B\}$;

$d_i := d_i - \delta$, $u^r_{0q} := u^r_{0q} - \delta$, $addvalue^2[i, q]\ (-\delta)$;

if $\delta = \Delta_B = u^r_{p',p'-1}$, **then** *split*[p′, b] from *B*;

list := (p′, list);

B : = [i, p′ - 1] and b = p′ - 1;

end;

The procedures can be extended to work for convex production costs and all convex costs with respective complexities: O(n log n log U) time for $U = \sum_{i=1}^{n} d_i$, and O(n^2 log U). For K products and K demand values at each period, the corresponding network includes O(nK) nodes and O(nK) arcs. A straightforward extension of the results yields O(nK log n) time algorithm for the linear cost model, an O(nK log n log U) time algorithm for the convex production cost model, and an O($(nK)^2$ log U) algorithm for the convex production, inventory and backorder costs model.

Acknowledgements:

The research of the first author was supported by the NSF Grants DMI-0085682 and DMI-0217359. The research of the second author was supported by NSF awards No. DMI-0085690 and DMI-0084857.

References:

Ahuja, R.K., T.L. Magnanti, and J.B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall.

Bitran, G.R., and H.H. Yanasse. 1982. Computational complexity of the capacitated lot size problem. *Management Science* **28**, 1174-1186.

Cormen, T.H., C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction to Algorithms*. Second Edition. MIT Press, Cambridge, MA.

Federgruen A., J. Meissner, and M. Tzur. Progressive integral heuristics for multi-item capacitated lot sizing problems. To appear in *Operations Research*.

Florian, M., J.K. Lenstra, and A.H.G. Rinnooy Kan. 1980. Deterministic production planning: Algorithms and complexity. *Management Science* **26**, 669-679.

Sedeño-Noda, A., J. Gutiérrez, B. Abdul-Jalbar, J. Sicilia. 2004. An O(T log T) algorithm for the dynamic lot size problem with limited storage and linear costs. *Computational Optimization and Applications* **28**, 311-323.

Wagner, H.M., and T.M. Whitin. 1958. Dynamic version of the economic lot size model. *Management Science* **5**, 89-96.

Tarjan, R.E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia, PA.