An Algorithm for Capacity Expansion in Local Access Networks

Andrew Coyle
Department of Engineering Science
University of Auckland
New Zealand
swcoyle@iprolink.co.nz

Abstract

Growing demand, innovations in telecommunication technologies, and the expansion and diversification of services offered have created a variety of design and expansion problems in telecommunication networks. This paper looks at one of these problems, the Local Access Network Expansion Problem (LANEP), in which existing cables and concentrators must be installed or upgraded to handle growing demand. The general problem can be modelled as an integer linear program, but this is known to be NP-hard and thus typical network sizes preclude the use of such methods. Various other solution methodologies and algorithms have been proposed, including a dynamic programming algorithm which is solvable in pseudo-polynomial time. This paper presents an extension to this algorithm which allows two of the major LANEP assumptions (contiguity and non-bifurcation) to be easily switched on or off. Translating this general LANEP algorithm into a real world application does create practical problems which prevents it from being implemented as is. These problems are discussed and current solutions under investigation are presented.

1. Introduction

Customer growth, increasing demands for old and new services, deregulation and the innovations in switching and transmission technologies are all placing pressure on telephone companies to upgrade and expand their networks. With over 50% of a telephone company's total investment in communication facilities[1] and with deregulation making the market more competitive these expansion decisions can have huge economic and strategic ramifications.

There are a variety of different problems which arise from communication networks, this paper deals with the Local Access Network Expansion Problem (LANEP). A local access network (LAN) is a tree where different nodes are connected to each other via an exchange located at the root of the tree. When a node wishes to send demand to another node in the tree, the traffic must be routed via the exchange. As demands grow, various links in the LAN become exhausted and network capacity needs to be expanded. In the past all expansion was done with the installation of copper cables, but recently technological advances have enabled electronic multiplexing and fibre optic transmission to be introduced as additional cost-effective options. Their introduction has however made expansion problems extremely complex as it presents the network planners with larger and larger numbers of expansion possibilities to distinguish between and compare. Various solution methods (both singly and combined) have been proposed, including iterative approaches, problem decompositions, Lagrangian relaxations with cuts, and dynamic programming. This paper focuses on a dynamic programming algorithm designed by Flippo, Koster, Kolen and van de Leensel[2], we propose an alteration of its mathematical formulation which is simpler while allowing some commonly made LANEP assumptions to be easily ignored if wanted.

The rest of the paper is organised as follows. Section 2 describes the problem in detail. Section 3 introduces the notation and discusses the dynamic programming algorithm created. Section 4 formally presents the dynamic programming algorithm. Section 5 discusses a practical problem which was encountered when trying to implement the algorithm and describes the solution to this problem currently under investigation.

2. Problem Description

The Local Access Network (LAN) is a network which connects customer nodes to the local exchange (switching center). Topologically most LANs have a tree structure with the exchange situated at the root and it is assumed that this structure will not change over the time horizon we are dealing with. Each customer node represents a collection (possibly hundreds or thousands) of individual customers, connected to this node via some other form of underlying network. The arcs in the LAN correspond to existing cables or sets of cables joined together in ducts. Demand at each customer node is generated when individual customers wish to send/receive traffic (such as voice or data) to another node in the LAN. This demand is measured by the required number of circuits from that node to the exchange (for example, normal phone conversation would require 2 circuits whereas high speed data transmission would require more) and is the sum total of all the individual customer circuit requirements. (Naturally a node which represents 100 telephones would not require 200 circuits to always be available as not all the phones will be used at the same time. Various calculations are typically applied to each node to generate the circuit demands such that enough circuits are provided to meet the typical expected load).

All communication to and from each customer node is via the switching center and the transmission of traffic from a customer node to the exchange can be done in one of two ways. It can either be passed up via the collection of existing arcs which form the unique path from this node back to the root of the tree, or it can be routed to a concentrator located somewhere in the tree. A concentrator is a device which compresses multiple incoming circuits onto a smaller number of outgoing circuits and sends these up to the exchange (for example multiple analogue circuits which are used for telephone conversations may be digitally multiplexed together onto a single line, or maybe even onto a fibre optic line). The concentrated circuits are either sent via a dedicated line which is not part of the tree, or by copper cable which was “stolen” from the tree, conditioned to handle the compressed circuits (for example installing repeaters on the cable) and now used as a dedicated line. It is usual to assume that if a concentrator steals copper cables from the tree, then the amount it steals is negligible, and thus does not need to be considered (this is a key assumption and one which we shall be referring back to later).

As customer demand grows (for example a new subdivision is built), the current cables and concentrators become exhausted and the network will need to be upgraded. This can be done by installing new concentrators, expanding existing cable capacities, and rerouting traffic through these new additions to the network. Thus the LAN expansion problem fundamentally becomes the economic trade-off between installing concentrators and cable expansion. In Balakrishnan, Magnanti, and Wong [3] they indicate that this problem is NP-complete.

While the problem can be modelled in a variety of ways, most models share a common set of assumptions which restrict the allowable expansion choices. These assumptions have arisen both through technological considerations and operational and management factors. These restrictions include the following (taken from [3]):

Assumption 1: Single-level concentration

Demand is concentrated at most once before it reaches the exchange.

Assumption 2: Non-bifurcated routing

Every node has its entire demand processed by a single concentrator or by the exchange (processed by the exchange means the node's demand is sent up the tree via the copper cables to the exchange).

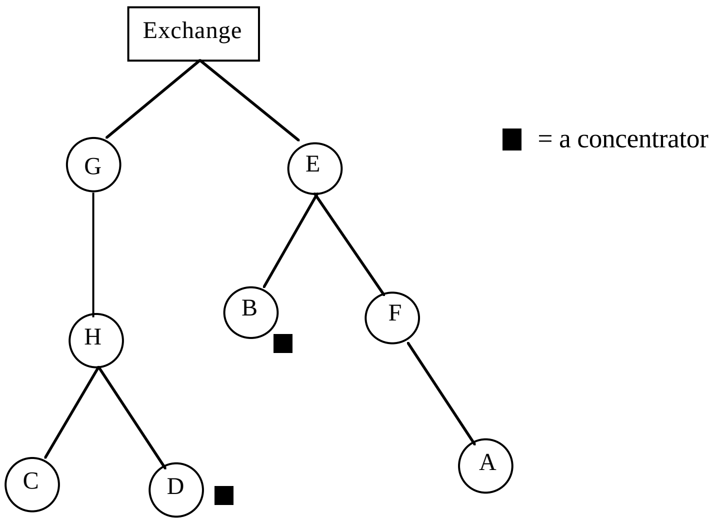
Assumption 3: Contiguity restriction

If node A is sending its demand to node B to be concentrated, then every node on the path from A to B also sends their demand to the concentrator at B. (note: it is assumed that there is always a concentrator installed at the exchange).

Assumption 4: Transmission cost for concentrated traffic

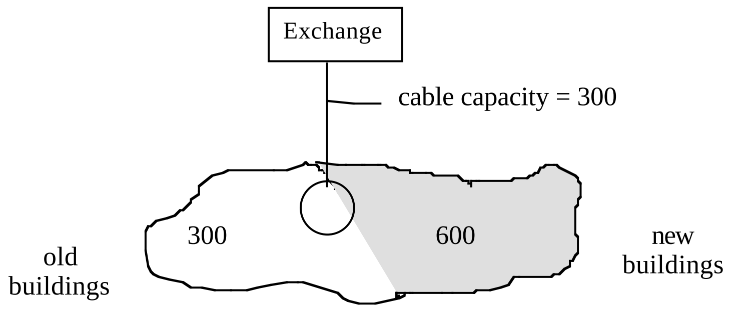
When sending concentrated traffic to the exchange, a concentrator either uses a negligible amount of existing cable capacity, or uses its own dedicated line

Assumption 1 reflects the economics of current local access network technologies - concentrating already concentrated circuits is not cost effective. Assumptions 2 and 3 are best understood by example. Take the following LAN



If node C sends its demand to the concentrator at D, then it has to send all of its demand there. It cannot send some of its demand to D and the rest up to the exchange (assumption 2). If node A sends its demand to the concentrator at B, then nodes E, F and B also have to send their demands to that concentrator (assumption 3).

Assumptions 2 and 3 are normally imposed for operational convenience as they simplify the representation, repair and maintenance of the networks. They can however force expansion plans which seem economically ill-considered. As an example of this, take the following LAN which services an area which has had some new buildings erected in it. Existing demand is shown in the blank region, the new demand in the shaded region.



The combined old and new demand will exhaust the current cable's capacity and thus the network needs to be upgraded. Assumptions 2 and 3 allow only two upgrade options: either the existing cable is expanded to handle 900 circuits (in this case the concentrator which is processing the nodes demand is the exchange), or a concentrator is installed at the node to process all the 900 circuits and the existing cable is disused. If there is a difference in cost between a concentrator which can handle 600 circuits and one which can handle 900 circuits then a cheaper option (but one which violates assumptions 2 and 3) would be to leave the old demand (300) using the existing cable capacity and install a concentrator solely to pick up the new demand. In the algorithm we present in section 3 and 4, neither assumption 2 nor 3 are required, but it is formulated in such a way that if operational convenience is believed to outweigh the economic downside then they may be easily "switched back on". (Note: as assumption 2 and 3 are fairly complementary we will combine their reference together and call them the "contiguity assumption").

Assumption 4 effectively says that if we decide to send some demand to a concentrator then that demand no longer needs to be considered by the network, it magically finds its way back to the exchange without interfering with any of the arc capacities. This greatly simplifies the solution process and is generally implemented due to the high compression ratios achieved by the concentrators (for example 1000 copper pairs for analogue transmission require only 11 copper cables to transmit the compressed signals [3]). It does however distort both the costs of concentrators which steal copper cable, and the true available capacity of the arcs. (This assumption is the one which causes the problem discussed in section 5).

A solution possibility of LANs worth noting is that of *backfeed*, or flow away from the exchange. This occurs when a node sends its demand for concentration to a node further away from the exchange than it is. This has the effect of freeing up cable higher up in the tree for use by other nodes.

Section 3. Notation and Algorithm Introduction

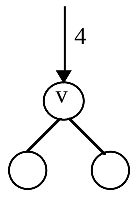
(Note: for ease of reference the notation and description below is derived from [2]).

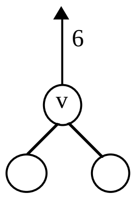
Let T = (V,E) be the tree on which the LANEP is defined, with V representing the set of nodes (V={0,…,n} with 0 being the root (exchange)) and E the set of edges (E = {1,…,n}). We number the nodes and edges in T in depth-first order so that node *v* and the edge above node *v* have the same numerical label. Let d_v be the number of children of node *v* in T, and let s_v^i be the ith child of node *v* (i = $1..d_v$). Let T[v,i] be a subtree composed of node *v* as the root, its first i children and all the successors of these children.

Let $K_v(r)$ represent the cost of using a concentrator at node *v* to process a load of *r*. Let $L_e(l)$ represent the cable costs of allowing a load of *l* to flow over edge *e*. Provided assumption 4 holds, this cost structure may be of any form, and given an *r* or *l* the values can be pre-calculated (the reader is directed to [2] for a detailed discussion of general and special cases of $K_v(r)$ and $L_v(l)$).

Before the formal description of the algorithm is given we give a brief and general explanation of how the heart of the algorithm works. Consider a (red) subtree with node *v* as the root and i of node *v*'s children (ie T[v,i]). Let us assume that we have available the optimal (minimal) costs of passing out (receiving in) *r* normal unconcentrated copper circuits through node *v* (a negative value of *r* indicates that we are receiving circuits in), and this value is stored in $\text{Cost}_{\text{red}}(r)$.

eg





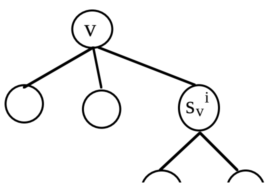
$\text{Cost}_{\text{red}}(-4)$ = \$100 $\text{Cost}_{\text{red}}(6)$ = \$50

Due to assumption 4, *r* represents normal unconcentrated copper circuit flow and completely ignores any concentrated flow.

A special case occurs when the red tree only has one node (*v*). If it is receiving in *r* circuits then it must be concentrating both these *r* circuits *and* node *v*'s demand (and sending them up to the exchange via the concentrator's dedicated line). Therefore the cost is $K_v(\text{demand}[v] + |r|)$. If it is sending out *r* circuits then it can send out at most node *v*'s demand (and thus incur no cost), and if it is sending out less than this then the remainder must be being concentrated at node *v*, giving a cost of $K_v(\text{demand}[v] - r)$.

Let us also assume that there exists a (blue) subtree with root s_v^i where s_v^i is the (i+1)th child of *v* but not included in the red subtree.

ie.



The blue subtree contains all the children of s_v^i, and all their successors (ie. T[s_v^i, $d_{s\,i}$]). Again, we have available the optimal costs of passing out (receiving in) *y* circuits through node s_v^i.
What the algorithm effectively does is take these two subtrees (red is T[v,i] and blue is T[s_v^i, $d_{s\,i}$]) and puts them together to create the "red-blue" subtree: T[v,i+1]. It uses the red and blue subtree optimal values to calculate the optimal values for the red-blue subtree (ie. the cost of passing out (receiving in) *r* circuits into the red-blue tree.). The value of $Cost_{red\text{-}blue}(r)$ is calculated via the following algorithm:

```
For r:= -∞ to ∞ do      {for all the possible number of circuits we can pass into/out of
                        v}
    begin
        Cost_red-blue(r):= ∞      {initialise}
        for y:= -∞ to ∞ do       {for all the possible number of circuits we can
                                 {pass into/out of s_v^i}
            begin
                tempValue:= Cost_red(r-y) + Cost_blue(y);
                if |y| > CapacityOfArc(s_v^i) then
                    *tempValue:= tempValue + (cost of upgrading
                                           arc s_v^i to handle y circuits)
                if tempValue < Cost_red-blue(r) then
                    Cost_red-blue(r):= tempValue
            end
    end
```

(* technically this statement is tempValue := tempValue + $L_v(y)$)

To illustrate how this merging works, take the following example. Assume we have already calculated the optimal costs of passing in/out *r* (*y*) circuits into/out of the red (blue) subtrees. These costs are:

Red Subtree

r	<(-10)	-10	-5	0	2	5	10	>10
$Cost_{red}(r)$	∞	160	135	125	70	20	0	∞

Blue Subtree

y	<(-5)	-5	0	3	5	10	>10
$Cost_{blue}(y)$	∞	110	105	100	50	0	∞

These values are now going to be used to create the red-blue subtree values. The following explanation steps through the calculations used for one of these entries ($Cost_{red\text{-}blue}(5)$). The other entries would be calculated in a similar fashion. $Cost_{red\text{-}blue}(5)$ represents the optimal cost of passing 5 circuits up from the red-blue subtree on arc *v*. There are a variety of ways we can merge the red and the blue subtrees together such that we are left with the red-blue subtree passing out 5 circuits. The algorithm identifies and enumerates every one of these, and then picks the cheapest one.

One way for the red-blue subtree to pass up 5 circuits would be for the red tree to pass out 10 circuits, 5 going up arc *v* and the other 5 going down arc s_v^i into the blue tree (figure 1 (a)). The cost of this is $\text{Cost}_{\text{red}}(10)$ + $\text{Cost}_{\text{blue}}(-5)$ + $L_{s\ i}(5)$ (for simplicity lets assume arc s_v^i has enough capacity to handle any flow and thus never needs to be upgraded), which gives a total cost for this solution of: \$0 + \$110 + \$0 = \$110.

Another way for the red-blue subtree to pass up 5 circuits would be for the red subtree to pass up 2 circuits and the blue tree to pass 3 up circuits which go straight through node *v* and onto arc *v* (figure 1 (b)). The cost of this solution is $\text{Cost}_{\text{red}}(2)$ + $\text{Cost}_{\text{blue}}(3)$ + $L_{s\ i}(3)$ = \$70 + \$100 + \$0 = \$170.

The red-blue subtree could also pass up 5 circuits by having the blue subtree pass out 10 circuits and the red tree grabbing 5 of these and leaving the remaining 5 to travel up arc *v* (figure 1(c)). The cost of this solution is $\text{Cost}_{\text{red}}(-5)$ + $\text{Cost}_{\text{blue}}(10)$ + $L_{s\ i}(10)$ = \$135 + \$0 + \$0 = \$135.

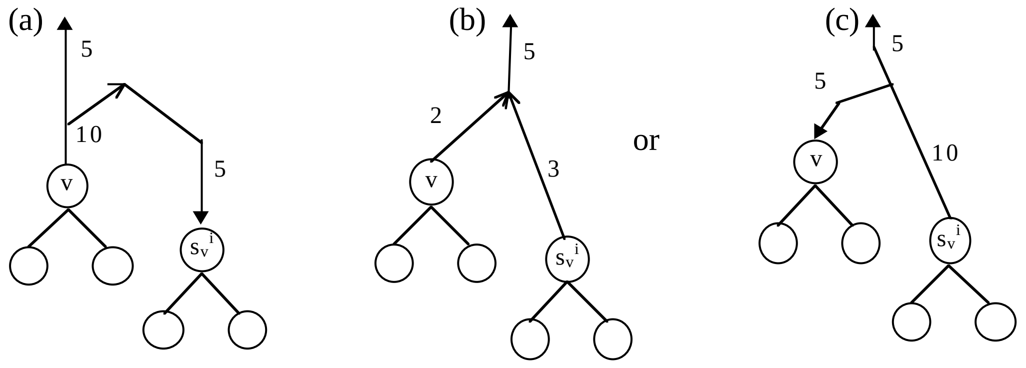


Figure 1

There are in fact 5 possible ways inflows and outflows from the red and blue subtrees can be combined which result in the red-blue subtree passing up 5 circuits. The remaining two possible solutions are:

$\text{Cost}_{\text{red}}(5)$ + $\text{Cost}_{\text{blue}}(0)$ + $L_{s\ i}(0)$ = \$20 + \$105 + \$0 = \$125

and

$\text{Cost}_{\text{red}}(0)$ + $\text{Cost}_{\text{blue}}(5)$ + $L_{s\ i}(5)$ = \$125 + \$50 + \$0 = \$175

The cheapest of all the solutions is chosen (in this case it was the first solution we looked at which had a cost of \$110) and this becomes the entry in the red-blue subtree table for $\text{Cost}_{\text{red-blue}}(5)$.

By an inductive proof it can be shown that every possible way for a flow of *r* to leave/enter the red-blue subtree is considered, and therefore the optimal values for the red-blue subtree will be created.

My algorithm works by taking the LAN tree and slowly building it up using red and blue subtrees, until finally the whole tree has been created. The optimal value for $\text{Cost}_{\text{whole-LAN-tree}}(0)$ gives the optimal solution for the LANEP.

Section 4. A Dynamic Programming Algorithm for LANEP

To help link section 3 with section 4 define f(v, i, r) = $\text{Cost}_{\text{tree}}(r)$ where tree = T[v, i].

DYNAMIC PROGRAMMING ALGORITHM FOR LANEP

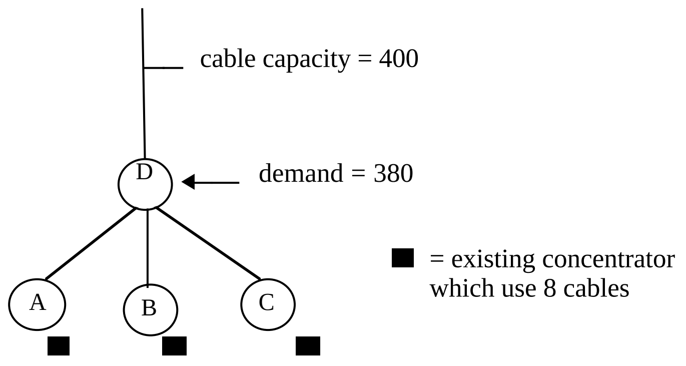
```
for all v = n downto 0 do
    begin
        {calculate the single node costs}
        for all r = 0 to demand[v] do
            f(v, 0, r) = $K_v$(demand[v] - r);
        for all r = (demand[v]+1) to ∞ do
            f(v, 0, r) = ∞;
        for all r = -∞ to -1 do
            f(v, 0, r) = $K_v$(demand[v] + |r|)
        for all i = 1 to $d_v$ do
            for r = -∞ to ∞ do
                begin
                for y = -∞ to ∞ do
                    begin
                    temp = f(v, i, r-y) + f($s_v^i$, $d_{s\ i}$ , y) + $L_{s\ i}$(y);

                    {cost contiguity and backfeed}
                    if (r<0) and (y<0) then
                        begin
                        temp = temp + backfeedCost;
                        if |y| < (|r| + demand[v]) then
                            temp = temp + contiguityCost;
                        end
                    else if (r>0) and (y<0) then
                        begin
                        temp = temp + backfeedCost +
                            contiguityCost;
                        end
                    else if (r>0) and (y>0) then
                        begin
                        if s<(y + demand[v]) then
                            temp = temp + contiguityCost;
                        end

                    if temp < f(v, i+1, r) then
                        f(v, i+1, r) = temp;
                    end
                end
    end
```

Observe that section 4 presents some additions to the heart of the algorithm given in section 3, namely the introduction of contiguity costs and backfeed costs. These are the mechanisms we discussed earlier by which the user is able to "switch" contiguity and backfeed "on or off". The variables backfeedCost and contiguityCost store the penalties for the current solution breaching the contiguity condition or allowing backfeed. If the user wishes to force contiguity or ban backfeed then the appropriate cost should be set to ∞, thus making the current unwanted solution so undesirable that it will never be picked. Likewise, if the user wishes to allow non-contiguity or backfeed, then they can set the appropriate cost to zero, thus imposing no penalty at all. The user can also effect a cost between the two extremes which would enable the algorithm to consider, but cost appropriately, solutions which were allowable but undesirable. As it is shown, the algorithm uses the same costs for all nodes but it would be trivial to extend this to allow node/arc specific contiguity and backfeed costs. It should be noted that if the user is *certain* that they will always/never want contiguity and/or backfeed then the algorithm can be made more efficient by removing these cost variables and simply altering the data ranges that the r and y loops run through.

For trees with a large number of nodes and/or bushy trees and/or trees with a large range of possible values of *r*, the number of variable instances of f(v, i, r) can grow large and thus memory restrictions could become a problem. Two observations can be made which reduce the amount of memory the algorithm requires. Firstly (using section 3 terminology), once a red and blue subtree have been combined together to form a red-blue subtree, the red and blue subtrees can be thrown away. As an extension of this it can be shown that an entire LANEP can be solved by storing only three subtrees at one time. Secondly, a LAN whose exchange node has multiple children can be split up into smaller sub-LANs, each containing the exchange, one (and only one) of the "exchange children" and all of that child's successors, and then solved independently. This decoupling is possible because the only path between a node in an exchange child subtree and a node in a separate exchange child subtree contains the exchange, and it would never be optimal to send data through the exchange and out again for the sole purpose of being sent back to the exchange. Splitting up the LAN into smaller sub-LANs reduces the number of nodes (v) and thus requires fewer instances of f(v, i, r).

Section 5. Extension

One of the major drawbacks with the algorithm above is that in practice assumption 4 is not valid. Assumption 4 effectively assumes that a concentrator will steal negligible amounts of the existing copper cables and therefore we can ignore any impact that concentrated traffic has on cable capacities. For LANs with huge cable sizes (very often in the thousands or tens of thousands) this may well be the case, but for exchanges with smaller cable capacities (like the ones we are dealing with), the accumulation of concentrators downstream can result in an "overflow" on an upstream cable. For example take the following LAN:



The concentrators at A ,B and C together use 24 copper cables, which when added to the 380 copper cables used by node D, exceed (overflow) the actual capacity of arc D.

In general, the cost of digging up a duct in order to lay more cable is a huge cost (possibly more expensive than installing a different concentrator downstream which has a higher concentration ratio), and thus we cannot simply recommend that all overflow arcs should have more cable laid.

Our proposed solution to this problem is to introduce a new state, *s*, which represents using *s* copper cables for concentrators. This expands our variable to become f(v, i, r, s) which holds the optimal cost of sending up *r* circuits using normal copper and *s* circuits using concentrator stolen copper in tree T[v,i]. When considering a solution, if (*r* + *s*) > capacityOfArc, then we need to upgrade the arc, and thus the algorithm has the digging costs added in for this solution. What needs to be investigated is how this additional state will increase the memory requirements. Another factor to be considered is that not all concentrators condition the copper cables in the same way and thus we may need to distinguish between the different types of concentrator copper cables (ie introduce a state t and u and v etc). If this is required then the state space could become very large indeed.

References

[1] Standard and Poor’s Industry Surveys (1992)

[2] O.Flippo, A.Koster, A.Kolen, and R.van de Leensel. “A Dynamic Programming Algorithm for the Local Access Network Expansion Problem”. *source unknown*

[3] A.Balakrishnan, T.Magnanti, and R.Wong. “A Decomposition Algorithm for Local Access Telecommunications Network Expansion Planning”. *Operations Research*, 43(1):58-76, 1995

[4] G.Cho and D.X.Shaw. "Limited column generation for local access telecommunication network design - formulations, algorithms, and implementation". Working Paper, January 1995.