Bond University
ePublications@bond

Information Technology papers | Bond Business School

12-1-2006

An Exact Algorithm for the Constant and Non-linear Cost Function Problem in Economic Synthesis Of Networks

Stephen J. Sugden
Bond University, ssugden@bond.edu.au

Jun Han
Bond University

Graham McMahon
Bond University

Follow this and additional works at: http://epublications.bond.edu.au/infotech_pubs

Recommended Citation
Stephen J. Sugden, Jun Han, and Graham McMahon. (2006) "An Exact Algorithm for the Constant and Non-linear Cost Function Problem in Economic Synthesis Of Networks" ,, .

http://epublications.bond.edu.au/infotech_pubs/60

This Journal Article is brought to you by the Bond Business School at ePublications@bond. It has been accepted for inclusion in Information Technology papers by an authorized administrator of ePublications@bond. For more information, please contact Bond University's Repository Coordinator.

An Exact Algorithm for the Constant and Non-linear Cost Function Problem in Economic Synthesis Of Networks

Jun Han, Graham McMahon and Stephen Sugden
School of Information Technology
Bond University

Abstract

In this paper we introduce a branch and bound algorithm for finding a tree solution of the minimum cost network flow problem with constant arc cost. We consider different situations such as unit and non-unit traffic demand. The entire searching process is interpreted and the methods used in different situations to prune the search tree are emphasized respectively. Empirical results indicate the efficiency of these techniques. While no algorithm exists for a piecewise-constant cost function problem, we provide a model for transforming the problem to one with constant arc costs. After the transformation, the piecewise constant cost function problem can be solved by most existing algorithms for the linear cost function problem.

1. Introduction

With the advent of the information age, there has been increased interest in the efficient design of communication networks. Considerable research interest has been shown in problems dealing with the economic synthesis of networks. The aim of this kind of design problem is to satisfy all traffic requirements at minimum cost between a set of nodes. The information required to formulate the problem is the traffic demand between each origin and destination (O-D) pair, and the cost function for carrying traffic on each arc between nodes *i* and *j*.

Many studies consider the cost function as a linear function of flow allocated on an arc [2][3][6][7][10], and the mathematical formulation of the problem can be as follows:

Minimize

$$\sum_{p=1}^{n} \sum_{q>p}^{n} \sum_{r} h_r^{pq} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} a_{ij,r}^{pq} \qquad (1)$$

S.T.

$$\sum_r h_r^{pq} = F^{pq} \qquad \forall p,q$$

$$f_{ij} = \sum_{p=1}^{n} \sum_{q>p}^{n} \sum_r a_{ij,r}^{pq} h_r^{pq} \qquad \forall i, j$$

$$0 \le f_{ij} \le f_{ij}^{\max} \qquad \forall i, j$$

$$0 \le h_r^{pq} \qquad \forall p,q,r$$

where

n is the number of nodes,

h_r^{pq} is the amount of traffic on route r between O-D pair $p-q$,

c_{ij} is the cost per unit flow on arc (i, j),

$a_{ij,r}^{pq}$ is 1 if the arc (i, j) exists on route r between nodes p,q; 0 otherwise,

F^{pq} is the total flow requirement between O-D pair $p-q$,

f_{ij} is the total flow on arc (i, j),

$f_{ij}^{\max}$ is the upper bound of the available capacity on arc (i, j).

There exists a substantial literature dealing with this problem, and many solution techniques have been applied to solve it, such as Genetic Algorithm [3][7][9], Simulated Annealing [10], Tabu Search [8], and Integrated GA-LP algorithm [2].

These algorithms are all heuristics. In this paper we present a branch and bound algorithm. The branch and bound algorithm is an exact algorithm and so guarantees finding the optimal solution.

An exact procedure, which has to enumerate all feasible solutions, is exponential in the number of nodes, and is thus not applicable to large size problems, so heuristics are more practical in this case. But developing exact algorithms is still essential. Exact algorithms are practical for small and medium size problems, especially in the pre-design of networks when their runtime is acceptable in practice. In addition, exact algorithms are extremely significant when being employed to evaluate the efficacy of heuristics. They provide the optimal solution which is the most convincing reference when testing the heuristics being designed.

In this paper we consider the case that the cost of building a certain arc is a constant but not a linear function of flow on the arc. Thus the objective function of the problem is as follows:

Minimize $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} b_{ij}$ (2)

S.T.

$$f_{ij} = \sum_{p=1}^{n} \sum_{q>p}^{n} a_{ij}^{pq} F^{pq} \qquad \forall i, j$$

$$0 \le f_{ij} \le f_{ij}^{\max} \qquad \forall i, j$$

where

n is the number of nodes,

c_{ij} is the cost of building arc (i, j),

b_{ij} is 1 if the arc (i, j) exists in the solution tree; 0 otherwise,

a_{ij}^{pq} is 1 if the arc (i, j) exists on the route between nodes p, q; 0 otherwise,

F^{pq} is the total flow requirement between O-D pair $p - q$,

f_{ij} is the total flow on arc (i, j),

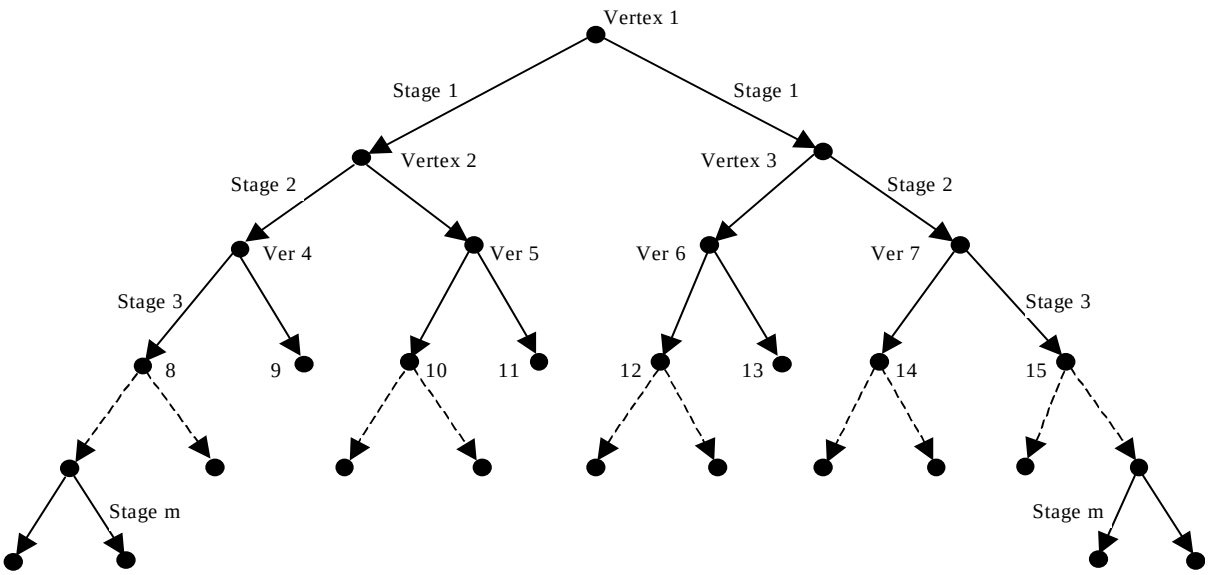
$f_{ij}^{\max}$ is the upper bound of the available capacity on arc (i, j).

From the next section, we will introduce a branch and bound algorithm for finding tree solution of the problem formulated in (2), which will be referred to as *constant arc cost problem*. In section 6, we will introduce a solution strategy for non-linear cost function problem.

2. The Search Tree

Since a tree spanning n nodes must have exactly n-1 arcs, we would like to consider the solution as a collection of n-1 arcs (for n-node problem) chosen from all candidate arcs, and would introduce an arc-oriented branch and bound method, branching by using an arc or not. A binary search tree is constructed as shown in Figure 1.

Figure 1 The search tree



We define an m-stage binary search tree for problem with n nodes, where, for the complete graph problem, m equals the number of pairs of nodes $\binom{n}{2} = n(n-1)/2$. We sort the arcs in the order of ascending arc cost and give each of them an arc number, from 1 to m, as identification. The branching within each stage corresponds to the decision of either adding a certain arc to the solution arc set or not. For example, in stage 3, we are deciding whether to choose arc 3 or not .

We define each left branch to represent choice of an arc, while a right branch corresponds to not choosing an arc. In Fig. 1, for example, at stage 1 and 2, traveling from Vertex1 to Vertex 2 then to Vertex 5 represents choosing arc 1 and not choosing arc 2; traveling from Vertex 1 to Vertex 3 then to Vertex 6 represents not choosing arc 1 but choosing arc 2.

It is clear that an optimal solution will be found after having visited all the vertices of this binary tree.

The process of searching for the optimal solution is actually the process of traveling along the branches of the search tree. The following rules are made for traversing the search tree:

1. Always start from the root of the tree (in Fig.1, Vertex 1).
2. Always go along the left branch first when descending.
3. Keep descending and backtracking when any of the following situations occurs:
    a. Current selected arcs make the solution infeasible.
    b. A feasible solution has been found.
    c. Total cost of selected arcs is more expensive than that of an existing solution.

4. After backtracking to a vertex along its left branch, go forward along its right branch.

   For example, in Fig.1, if go backward from Vertex 4 to Vertex 2, then go to Vertex 5.

5. After backtracking to a vertex along its right branch, go backward to its parent vertex.

For example, in Fig.1, if going backward from Vertex 5 to Vertex 2, then continue going back to Vertex 1.

If these rules are followed, the search procedure will finally come back to the start point from its right branch and cannot go farther anymore (in Figure 1, from Vertex 3 to Vertex 1), then all feasible solutions have been checked and the optimal solution is found.

3. Pruning The Search tree

Besides the routines described in section 2.2, we now introduce other techniques for pruning the search tree and reducing the steps in the search process.

3.1 SMST Bound

Each vertex in the search tree represents a sub-problem generated from the original one. The difference between a sub-problem and the whole problem is that, in a sub-problem, some arcs are not to be used while some other arcs must be included in the solution. If the capacity constraint is relaxed, the sub-problem is quite similar to the traditional minimum spanning tree (MST) problem. Here, we refer to the capacity constraint relaxed sub-problem as the SMST problem.

We introduce a Modified Kruskal's Algorithm for the SMST problem.

The Modified Kruskal's Algorithm starts from an empty solution arc set. Firstly, arcs that are not to be used are eliminated from the whole candidate arc set. Secondly, arcs that must be used are included in the solution arc set. Such arcs will not produce cycles, since at each vertex in the search tree, it is always checked whether a cycle is made after the corresponding arc is added to the solution. Next, the cheapest arc in the candidate set is chosen, and the feasibility after it is included in the solution is checked. If inclusion of the arc creates a cycle, it is abandoned. If it no cycle is formed, then the arc is included in the solution arc set permanently. Then the second cheapest arc is considered, and this procedure is repeated until a complete tree spanning all nodes is constructed.

Theorem 1 The solution found by the Modified Kruskal's Algorithm is the optimal solution for the SMST problem.

Proof To prove the optimality of this solution, the method of "reduction to absurdity" is used.

Assume that the solution tree constructed by the above procedure is not a minimum spanning tree. Then, the total cost of the "actual" optimal solution must be less than the cost of the current tree. This implies that at least

one arc must be replaced by another cheaper arc, which forms a feasible solution. Suppose arc (i, j) is in the current solution but not in the optimal one, and there exists another arc (k,l), which is cheaper than arc (i, j), in the optimal solution but not in the current one. Because the Modified Kruskal's Algorithm began with the cheapest arc, and second cheapest one etc., arc (k,l) must have been tested before arc (i, j). Since the feasibility is always checked, arc (k,l) not being in the current solution means it would have created a cycle, had it been included. Therefore, it is impossible that an arc (k,l), which is not included in the current solution and is cheaper than arc (i, j), can be in the optimal solution.

It can be deduced that, there exists no arc, which can be used to replace any arc in the current solution to form a better solution. Therefore, the current solution found by the Modified Kruskal's Algorithm is the optimal one. This completes the proof.

Since in the search tree, a certain vertex represents that some arcs have been chosen and some other arcs have been eliminated, we can use SMST as a bound. At each vertex in the search tree, in addition to the feasibility checks, the current SMST is constructed and its total cost is calculated. If it is more expensive than the cost of a solution already found, the sub-tree rooted at the current vertex in the search tree is pruned.

3.2 Predicting Violation of Capacity Constraint

In addition to the SMST bound, an efficient feasibility check procedure is essential for pruning the search tree. When analyzing the above search routine, we found some shortcomings. When a cheaper arc, which would have a smaller arc number and would possess a higher stage in the search tree, has a very low flow capacity, it will often spend a lot of time searching on the sub-tree under this stage while no feasible solution exists. For example, in Fig.1, when descending and reaching Vertex 8 (arcs 1, 2 and 3 have been chosen and make solution feasible at this moment). If arc 3 happens to have a very low capacity, then a lot of time might be spent visiting the descendants of Vertex 8.

While the traffic demand between each O-D pair could be equal or not equal, we employ different methods to prune the search tree quickly in these two situations.

3.2.1 Feasibility Predicting Method for Problem with Unit Traffic Demand

This is the case when the traffic demand between each O-D pair is equal, and we can just consider the traffic demand between each O-D pair to be 1 unit. While there is traffic demand between every pair of nodes, the flow on an arc can be calculated by just multiplying the number of nodes on one side of the arc and the number of nodes on the other side of the arc. For any arc (i, j) in the solution tree, let V_1 be the maximal sub-tree

containing node i but not containing node j; let V_2 be the maximal sub-tree containing node j but not containing node i. Let NL be the number of nodes in V_1, NR be the number of nodes in V_2, then the flow on arc (i, j) must be $NL \times NR$ units.

If arc (i, j) is in a complete feasible solution tree of an N node problem, then the sum of NL and NR must be equal to N. That is, when $NL = 1$, NR must be $N - 1$; when $NL = 2$, NR must be $N - 2$, and accordingly. Therefore, depending on the topology of the solution tree, the flow on arc (i, j) may be from $1 \times (N-1) = N - 1$ units to $(N/2) \times (N/2) = N^2/4$ units (when N even); or from $N - 1$ units to $((N+1)/2) \times ((N-1)/2) = (N^2 - 1)/4$ units (when N is odd).

Let $N_{\max}$ be the greater of NL and NR, and $N_{\min}$ the smaller. Then the possible combinations of $N_{\max}$ and $N_{\min}$, and the value of flow on arc (i, j) is as shown in Table 1.

It can be shown that, in Table 1, the value of flow (columns 3 and 6) is increasing from the top to the bottom of the table. We can see from Table 1 that, if arc (i, j) has been used to construct a feasible solution tree, and its $N_{\min}$ is m, then its flow capacity must be equal to or greater then m(N-m). Moreover, if arc (i, j) is to be chosen to construct a feasible solution tree and its flow capacity is T, then, when m(N-m)<= T <(m+1)(N-m-1), the arc's $N_{\min}$ must be less than or equal to m, since otherwise, there will be an overflow on the arc. Here we name the number m the *Link Degree* of arc (i, j). To form a complete feasible solution, the Link Degree of arc (i, j) is actually the maximum number of nodes that can be included in the smaller one between the two end node sets (V_1 and V_2) of the arc.

Before the search procedure begins, the Link Degree value for every arc is stored. At every step of the

Table 1 Possible Combinations of $N_{\max}$ and $N_{\min}$

N	is	even	N	is	odd
$N_{\min}$	$N_{\max}$	*Flow*	$N_{\min}$	$N_{\max}$	*Flow*
1	N-1	N-1	1	N-1	N-1
2	N-2	2(N-2)	2	N-2	2(N-2)
…	…	…	…	…	…
m	N-m	m(N-m)	m	N-m	m(N-m)
m+1	N-(m+1)	(m+1)(N-(m+1))	m+1	N-(m+1)	(m+1)(N-(m+1))
…	…	…	…	…	…
N/2	N/2	N*N/4	(N-1)/2	(N+1)/2	(N*N-1)/4

search, even the current partial solution is feasible, the procedure checks if any arc's current NL and NR are both greater than its Link Degree. If so, the procedure performs backtracking and the subtree rooted at the current vertex in the search tree is pruned, since it will sooner or later cause an overflow if more arcs are added to the solution.

Comparing the Link Degree and the current $N_{\min}$ can predict whether an arc will overflow even if it hasn't yet been connected to some other separated arcs or sub-trees and hasn't yet overflowed. Implementing this technique will prune the search tree far more quickly.

3.2.2 Feasibility Predicting Method for Problem with Non-Unit Traffic Demand

When the traffic demand between each O-D pair is not equal, we use the following method to calculate the possible minimum flow on an arc.

At a certain stage of the search, let F_c be the current traffic flow on arc (i, j), write V_l ($|V_l| = L$) for the end node set of arc (i, j) containing node i; V_r ($|V_r| = R$) for the other end node set of arc (i, j) containing node j, and V_s ($|V_s| = S$) for the set of nodes not having been connected to the arc. Every node m in set V_s will be eventually connected to arc (i, j), and will be included in either V_l or V_r. Let FL_m be the total traffic demand between node m and all nodes in V_l, FR_m be the total traffic demand between node m and all nodes in V_r, that is,

$$FL_m = \sum_{I=1}^{L} Demand(m, V_l(i))$$

$$FR_m = \sum_{I=1}^{R} Demand(m, V_r(i)).$$

Then the minimum flow increase on arc (i, j) after connecting node m to the arc will be either FL_m (when V_l stays unchanged) or FR_m (when V_r stays unchanged). If $F_{\min}^{m}$ be the smaller of FL_m and FR_m, then the minimum flow increase on arc (i, j) after connecting all nodes in V_s to the arc is

$$F_{MIN} = \sum_{j=1}^{S} F_{\min}^{V_s(j)}.$$

Let F_t be the minimum total flow on arc (i, j) after connecting all separated nodes to the arc, then $F_t = F_C + F_{MIN}$. If F_t is greater than the capacity of arc (i, j), we know that the arc must encounter overflow if a complete solution is formed.

At each stage of the search, F_t is computed for each arc, and if this exceeds capacity, the procedure will perform backtracking. Here, F_t works like a bound and helps the procedure prune the search tree much faster.

The F_t bound can be made even tighter if the possible minimum flow on an arc is calculated based on every separated sub-tree and not on a single node. We define V_s' ($|V_s'| = S'$) to be the set of separated sub-trees (including separated single nodes) of arc (i, j), $V_s'^k$ ($|V_s'^k| = S_k'$) be the set of nodes contained in the kth sub-tree of V_s'. Let FL_k' be the total traffic demand between all nodes in $V_s'^k$ and all nodes in V_l, FR_k' be the total traffic demand between all nodes in $V_s'^k$ and all nodes in V_r, that is,

$$FL_k' = \sum_{j=1}^{S_k'} \sum_{I=1}^{L} Demand(V_s'^k(j), V_l(i))$$

$$FR_k' = \sum_{j=1}^{S_k'} \sum_{I=1}^{L} Demand(V_s'^k(j), V_r(i))$$

Then the minimum flow increase on arc (i, j) after connecting the kth sub-tree to the arc will be either FL_k' or FR_k'. Let $F_{\min}'^k$ be the smaller of FL_k' and FR_k', then the minimum flow increased on arc (i, j) after connecting all sub-trees V_s' to the arc is

$$F_{MIN}' = \sum_{k=1}^{S'} F_{\min}'^k$$

Regarding each node m in sub-tree $V_s'^k$, since $F_{\min}^m$ is the smaller of FL_m and FR_m, that is, $FL_m >= F_{\min}^m$, $FR_m >= F_{\min}^m$, then for all nodes in sub-tree $V_s'^k$, we have,

$$FL_k' = \sum_{j=1}^{S_k'} FL_{V_s'^k(j)} >= \sum_{j=1}^{S_k'} F_{\min}^{V_s'^k(j)}, \quad FR_k' = \sum_{j=1}^{S_k'} FR_{V_s'^k(j)} >= \sum_{j=1}^{S_k'} F_{\min}^{V_s'^k(j)}.$$

Since $F_{\min}'^k$ is either FL_k' or FR_k', we have $F_{\min}'^k >= \sum_{j=1}^{S_k'} F_{\min}^{V_s'^k(j)}$.

Therefore, for all nodes in V_s, $F'_{MIN} = \sum_{k=1}^{S'} F'^{k}_{\min} >= \sum_{k=1}^{S'} \sum_{j=1}^{S'_k} F_{\min}^{V'^{k}_{s}(j)} = \sum_{j=1}^{S} F_{\min}^{V_s(j)} = F_{MIN}$.

After F_{MIN} is replaced by F'_{MIN}, we get a tighter bound $F'_t = F_c + F'_{MIN}$.

4. The Algorithm

In this section, the main search procedure of the algorithm is described. The programming language used here is Object Pascal.

4.1 Data Structures and Functions

The authors consider the following to be the best data structure for an arc-oriented branch and bound search.

Let N be the number of nodes, M = N(N-1)/2 be the total number of candidate arcs.

First, the arcs are sorted in the order of ascending arc cost and each given an arc number, 1 to m, as identification. These candidate arcs are stored in an array Arcs [1..M], and the first element Arcs [1] stores the record of the cheapest arc with arc number “1”, the second element Arcs [2] stores the record of the second cheapest arc with arc number “2”, etc.

Next, we define “SolutionBuffer”, which stores the arcs (identified by arc number) that have been chosen to form a potential feasible solution, as an array [1..N-1] of integer. For example, if at a certain step “SolutionBuffer” equals [1,3,7,0,...,0], it means arcs 1, 3, and 7 have be chosen. “Solution”, which is the same data type with “SolutionBuffer”, stores the best feasible solution already found.

An integer “FlagSolve” is used to store the total number of arcs already been added to “SolutionBuffer” at the current searching step, e.g., if “SolutionBuffer” is [1,3,7,0,...,0], then FlagSolve equals 3.

The following are brief descriptions of the functions called by the main searching procedure:

Function CostCheck (i: integer): Boolean. Returns true if and only if the total cost of current selected arcs is less than the cost of the best solution already found after adding arc i to the solution set.

Function TreeCheck (i: integer): Boolean. Returns true if and only if the inclusion of arc i in the solution tree would not create a cycle.

Function CapacityCheck (i: integer): Boolean. Returns true if and only if the current or future flow on each arc observes the capacity constraint after using arc i.

Function SMSTCheck (i: integer): Boolean. Returns true if and only if the SMST is cheaper than the best solution already found.

Function IniSolve: Integer. Returns the cost of an initial feasible solution found by a greed method.

4.2 Main Search Procedure

The main procedure for traversing the search tree is shown in Figure 2.

Integer “i” is the arc number of current processing arc; “TotalCost” is the cost of the best solution already found; “TempCost” is the cost of current partial solution. “TotalCost” and “TempCost” are global variables while “i” is a local variable.

The numbers in brackets are used to refer to the codes in the same line with these numbers respectively. For the sake of brevity, some variable assignments are omitted here.

```
Procedure MainSearching;
Var
i: Integer;
begin
TotalCost:=IniSolve; FlagSolve:=0; i:=1;        (1)
while (flagsolve>0) or (i<M-N+2) do begin       (2)
  if CostCheck(i) then begin                    (3)
    if TreeCheck(i) then begin            (4)
          if SMSTCheck(i) then begin      (5)
           if CapacityCheck(i) then begin       (6)
         Flagsolve:=Flagsolve+1;          (7)
         SolutionBuffer[FlagSolve]:=i;    (8)
         if Flagsolve=N-1 then begin      (9)
                     TotalCost:=TempCost;     (10)
            Solution:=SolutionBuffer;(11)
            i:=SolutionBuffer[N-2]; (12)
                     i:=i+1;          (13)
                     FlagSolve:=NN-3;         (14)
                   end
         else i:=i+1;     // Flagsolve<>N-1   (15)
        end
            else i:=i+1;  // CapacityCheck=False   (16)
      end
          else begin     //SMSTBoundCheck=False   (17)
             if FlsgSolve=0 then break;   (18)
             i:=SolutionBuffer[FlagSolve];     (19)
         i:=i+1;                          (20)
         FlagSolve:=FlagSolve-1;          (21)
      end;
     end
    else i:=i+1;          // TreeCheck=False       (22)
   end
  else begin   // CostCheck=False             (23)
           if FlagSolve=0 then break;          (24)
           i:=SolutionBuffer[FlagSolve];       (25)
      i:=i+1;                          (26)
      FlagSolve:=FlagSolve-1;          (27)
    end;
 end;
end;
```

Figure 2 Main Search Procedure

In (1), “TotalCost” is assigned the value of the cost of an initial solution. It works as the upper bound of the optimal solution and will be improved whenever a cheaper solution is found. Since the procedure starts from an empty solution arc set, the initial value of “FlagSolve” is 0. The arc number i is assigned 1 at the beginning, as in the first stage the procedure is considering arc 1, which is the cheapest arc of all.

As mentioned in section 4.1, the program stops when backtracking from the right branch of vertex 1. At this moment, FlagSolve equals 0. Also, to ensure that the complete search tree has been traversed, i should equal to M, so that the right most vertex in the search tree has been reached. Therefore, the termination condition is FlagSolve=0 and $i =$ M. Actually $i \geq$ M-N+2 is used instead of $i =$ M, since the number of candidate arcs will be N-2 when the first chosen arc is arc M-N+2, and there need to be at least N-1 arcs to form a solution. In (2), the condition of the “while” loop is FlagSolve>0 or i <M-N+2, which is the inverse of FlagSolve=0 and i >=M-N+2 (“FlagSolve” can never be less than 0 in the procedure).

The following are interoperations of the processing after each function call.

- Function “CostCheck(i)”

In (3), function “CostCheck” is called, where the procedure checks if the total cost of current selected arcs and arc i is less than the upper bound; If so, the procedure goes on processing cycle check (4). Otherwise (23), backtracking in the search tree is performed. This bound is helpful especially when there is a jump in the costs of two successive arcs.

(24) works when the procedure comes to processing large weight arcs. FlagSolve=0 implies that arc i is the only arc selected, and all arcs which are cheaper than arc i are decided not to be used. If arc i is more expensive than the upper bound, any other arc is also more expensive than the upper bound. Then the procedure will finally face the termination condition, FlagSolve=0 and i >=M-N+2, without finding any cheaper solution, so the search is stopped now.

If the above is not the case, the procedure will perform backtracking. The reason why the rest of the arcs are not going to be considered is simply that, they are all more expensive than arc i. Eliminating arc i and adding any of them will also cause the “CostCheck” function to return false.

Let j be the last arc already selected (SolutionBuffer[FlagSolve]). Since no arc between j and i is chosen, the backtracking will be all along right branches till reaching the stage corresponding to arc j in the search tree. (25) retrieves arc j, (26) gets the next arc of j and FalgSolve is decreases by 1 in (27).

- Function “TreeCheck(i)”

In (4), function “TreeCheck(i)” is called, where the procedure tests if a cycle is made after adding arc i to the solution. If the partial solution is still a tree, function returns true and the SMST bound is checked in (5). If arc i makes a cycle to the partial solution, function “TreeCheck(i)” returns false, and in (22), i is increased by 1, which means the next arc is going to be considered.

- Function “SMSTCheck(i)”

In (5), function “SMSTCheck(i)” is called, where the procedure tests if the SMST cost, after arc i is added to the current partial solution, is less than the global upper bound. If so, the procedure goes on processing the capacity check (6); otherwise (17), a backtracking is performed.

Let C_p be the cost of the current SMST, it can be proved that, in the current partial solution, replacing arc i with any other expensive arc will lead to a SMST that must cost more than C_p. This is the reason why the next arc of arc i needs not to be considered and the traversal on the search tree can be backwards.

(18) ~ (21) function the same as (24) ~ (27).

- Function “CapacityCheck(i)”

In (6), function “CapacityCheck” is called. If adding arc i to the solution causes a violation of capacity constraint, arc i is discarded and the next one, arc i +1 (16), is tested.

If the capacity constraint is still observed after adding arc i to the solution, arc i is made permanent. “FlagSolve” is increased by 1 (7), and arc i is stored in the corresponding position in “SolutionBuffer” (8).

- Finding a Solution

(9) judges if a complete solution has been found. FlagSolve=N-1 tells there are N-1 arcs in the solution arc set, and since every arc has “passed” the “Tree Check” and “Capacity Check”, these arcs must form a feasible complete solution. Furthermore, the true value of function “CostCheck” shows that this solution cost is less than the best solution cost found before. (10) assigns the new upper bound, and (11) updates the best solution.

Next, a backtracking is performed. Decreasing “FlagSolve” by 1 is not helpful, because this will cause only one arc, which is arc i actually, to be replaced by an arc that is more expensive. Therefore, the traversal backtracks two stages to look for a potential better solution. (12) retrieves the second last arc in “SolutionBuffer” and (14) decreases “FlagSolve” by 2.

5. Computational Experience

A number of sets of instances generated randomly are used to test the branch and bound algorithm. Each random variable is evenly distributed within a certain range. The computations were performed on a machine with Pentium II 400MHz CPU, 128MB RAM, running the Microsoft Windows 95 operating system.

Table 2 shows the results of one set of 20-node problems with unit traffic demand. Method I is our branch and bound algorithm using direct feasibility check (calculating the flow already allocated on each arc). In method II, we use Link Degree to predict the feasibility of a potential solution in unit traffic demand problems. From Table 2 we can see that method II reduced the total number of search steps by 99 percent (one "step" corresponds to traversing one arc in the search tree). In addition, Method II can solve up to 100-node problems while Method I cannot. Table 3 shows the results for one set of 50-node problems and one set of 100-node problems using method II.

We employ F_{MIN} in Method III to predict the flow on each arc for non-unit traffic demand problems, and we use F'_{MIN} instead of F_{MIN} in Method IV. The performance of Method III and Method IV when solving a set of 50-node problems and a set of 100-node problems are shown in Table 4 and Table 5 respectively. Table 4 and Table 5 show that Method III and Method IV can solve up to 100-node non-unit demand problems and Method IV run less searching steps than Method III in many cases.

All the test problems and solutions can be accessed at http://www.geocities.com/ilhll/mincostflow.html.

Table 2 Results for 20-node unit demand problems

Problem file Name	Method I Running Steps	Method II Running steps	Percentage of reduced steps
P20-1.txt	123766	216	99.825%
P20-2.txt	6445	655	89.837%
P20-3.txt	10890072	215	99.998%
P20-4.txt	9598	83	99.135%
P20-5.txt	4865529	423	99.991%
P20-6.txt	25087936	71	99.999%
P20-7.txt	4861602	237	99.995%
P20-8.txt	4364835	69	99.998%
P20-9.txt	10352988	1452	99.986%
P20-10.txt	612423	936	99.847%

Table 3 Results for 50-node and 100-node unit demand problems using Method II

50-node problem			100-node problem		
Problem File Name	Running Steps	Running Time (Seconds)	Problem File Name	Running Steps	Running Time (Seconds)
P50e-1.txt	468	1	P100e-1.txt	54773	11
P50e-2.txt	31888	3	P100e-2.txt	3119924	399
P50e-3.txt	22384	2	P100e-3.txt	488953	50
P50e-4.txt	59882	6	P100e-4.txt	99842	9
P50e-5.txt	8531	1	P100e-5.txt	6093068	265
P50e-6.txt	663497	72	P100e-6.txt	43912	25
P50e-7.txt	1303	1	P100e-7.txt	1702974	559
P50e-8.txt	84144	5	P100e-8.txt	155402	38
P50e-9.txt	28962	2	P100e-9.txt	4025	2
P50e-10.txt	17184	1	P100e-10.txt	216873	43

Table 4 Results for 50-node non-unit demand problems

	Method III		Method IV	
Problem File Name	Running Steps	Running Time (Seconds)	Running Steps	Running Time (Seconds)
P50g-1.txt	17493	2	16582	2
P50g-2.txt	1116	1	1116	1
P50g-3.txt	19202	4	18657	4
P50g-4.txt	31474	9	31218	10
P50g-5.txt	243984	65	242895	57
P50g-6.txt	4191	1	4191	1
P50g-7.txt	7500	2	7500	2
P50g-8.txt	110037	20	9589	19
P50g-9.txt	23034	4	22734	5
P50g-10.txt	8032	1	8021	1

Table 5 Results for 100-node non-unit demand problems

	Method III		Method IV	
Problem File Name	Running Steps	Running Time (Seconds)	Running Steps	Running Time (Seconds)
P100g-1.txt	169474	163	169102	163
P100g-2.txt	279842	407	276265	410
P100g-3.txt	156281	140	156281	146
P100g-4.txt	188611	160	188611	167
P100g-5.txt	3312304	720	3203941	681
P100g-6.txt	98158	182	98158	191
P100g-7.txt	217784	400	217356	398
P100g-8.txt	140683	163	140683	172
P100g-9.txt	628903	700	627798	710
P100g-10.txt	5804	8	5804	8

6. Non-Linear Cost Function Problem

For problems with non-linear cost function, we would like to study the case that the cost function of flow is a step function (piecewise constant), since we consider a step function embodies the primary characteristic of realistic communication networks, and could be a foundation for studying more complex and realistic network models.

Let K_{ij} be the total number of steps of the cost function $c(f_{ij})$ of arc (i, j), c_{ij} be the variable representing cost and f_{ij} be the variable representing flow quantity; then, as shown in Figure 3,

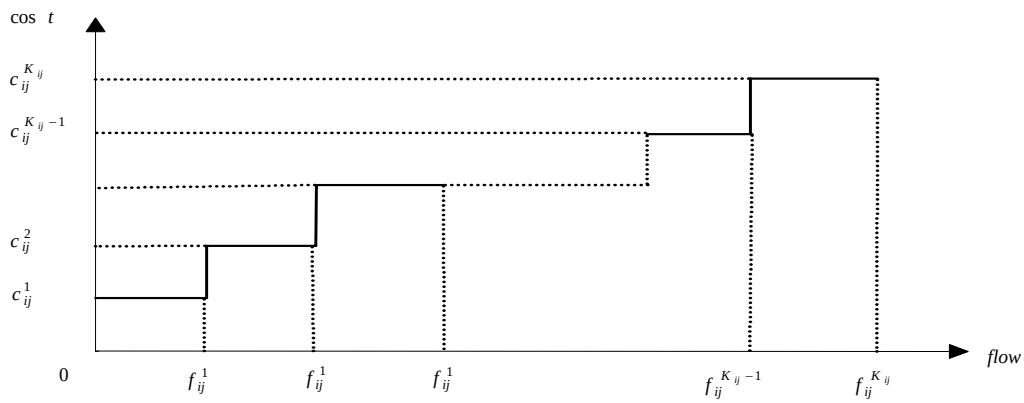


Figure 3

$$c(f_{ij}) = c_{ij}^{k_{ij}} \qquad \text{when} \qquad f_{ij}^{k_{ij}-1} < f_{ij} \le f_{ij}^{k_{ij}}$$

where $1 \le k_{ij} \le K_{ij}$, $f_{ij}^{0} = 0$ and $c(0) = 0$.

Therefore, the formulation of the problem will be:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c(f_{ij}) \qquad (3)$$

S.T

$$f_{ij} = \sum_{p=1}^{n} \sum_{q>p}^{n} \sum_{r} a_{ij,r}^{pq} h_r^{pq} \qquad \forall i, j$$

$$\sum_{r} h_r^{pq} = F^{pq} \qquad \forall p, q$$

$$0 \le f_{ij} \le f_{ij}^{K_{ij}} \qquad \forall i, j$$

$$0 \le h_r^{pq} \quad \forall p, q, r$$

where all variables are similarly defined as in Formulation (1).

4.1 A Simple Case

We first consider an arc (i, j), the cost function of which is shown in Figure 4. This is a simple case of Formula (3), which has only two steps.

We can see from the figure that when flow quantity is between 0 and 100, the cost of carrying traffic is 5, and when the flow quantity is between 100 and 150, the cost will be 14; the flow capacity of arc (i, j) is 150. Figure 5 shows an attempt at representing the single arc (i, j) by two "one step" arcs from i to j, A and B.

Arc A can carry traffic up to 100 with the cost value of 5 and arc B can carry 50 and costs 9. Any algorithm which chooses cheaper arcs first will choose arc A when allocating traffic less than 100 from i to j. When the flow requirement is more than 100, both arcs A and B will be chosen and the cost will be 14, but the total flow quantity cannot exceed 150.

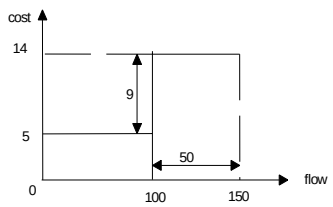


Figure 4

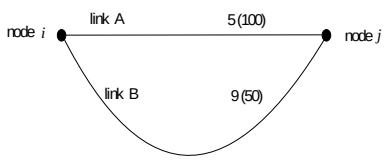


Figure 5

So we have successfully represented arc (i, j) with a two-step cost function by two arcs A and B, where each has a constant cost and a capacity limit.

4.2 A More Complicated Case

The example in the last section is a rather special case. A problem arises if we consider the example of Figure 6.

In Figure 6-a, since the second step is less than the first one, if we use the same transformation as in the last example, we will have 3 as the cost of arc B. Therefore, a routing algorithm will choose arc B when the traffic demand is less than 50, and Figure 6-b will not be a correct interpretation of Figure 6-a.

We can avoid this problem if we use the total function values instead of step values as the costs of arc A and B, and we use the total flow capacity of arc (i, j) as the capacity of arc B, as shown in Figure 6-c.

If the traffic is less than 100, arc A is chosen, and if the traffic is more than 100 and less than 150, arc B will be chosen (we never use both of them to carry traffic less than 150, since that will cost 5 + 8 = 13).

But there is still one problem. Arcs A and B can together carry 250 units of traffic from *i* to *j*, and the cost is 13. This is definitely not an accurate model since it adds a third step to our original cost function.

To avoid this, we must restrict the total flow capacity from *i* to *j*. If we consider arcs A and B as two parallel connecting arcs, we can restrict the flow quantity by adding another series arc between *i* and *j*, as shown in Figure 6-d.

The combination of arcs A, B and C has exactly the same behavior as arc *(i, j)*. Arc A has a constant cost of 5 and flow capacity of 100, while arc B has cost 8 and arc C costs 0 when carrying up to 150 units of flow.

4.3 General Model

A general cost function with two steps is shown in Figure 7-a, and Figure 7-b shows how three arcs A, B and C are used to replace arc *(i, j)*, and have exactly the same behavior when considering cost and flow from *i* to *j*.

Furthermore, when the cost function of arc *(i, j)* has three steps, we can also add a parallel connecting arc D and a serial connecting arc E to achieve our goal. Figure 8 is the expression of this method.

It can be deduced that an arc *(i, j)*, which has a $n(n \geq 2)$ step cost function of traffic, can be replaced by $(2n-1)$ arcs with constant cost when we add $(n-1)$ nodes between nodes *i* and *j*, as shown in Figure 9.

We often wish to deal with simple graphs, and we can make Figure 9 a simple graph by adding an extra node on arc B, and separate it into two arcs B and B*, as shown in Figure 10.

Thus, the piecewise constant cost function problem, formulated by Formula (3), has been successfully transformed to constant arc cost problem, and its objective function can be exactly the same as Formula (2):

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij} b_{ij}$$

4.4 Computational Experience

After being transformed to a constant arc cost problem, the piecewise constant cost function problem can be solved by all algorithms mentioned in Section 1, as well as our branch and bound algorithm. Table 6 show the results when solving some randomly generated instances by the branch and bound algorithm. All random variables for the parameters in the problems follow even distribution within a certain range. The computations were performed on a machine with Pentium II 400MHz CPU, 128MB RAM, running the Microsoft Windows 95 operating system.

The problems in Table 6 are 10-node problems with full density. That is, there is a candidate arc between each pair of nodes and the total number of candidate arcs is 10(10-1)/2 = 45. The cost of each arc follows a 2-step cost function. Therefore, two dummy nodes are inserted between each pair of nodes, and the total number of nodes of the new problem is 10 + 45*2 = 100. According to our model, each arc is replaced by four arcs and the total number of candidate arcs in the new problem is 45*4 = 180, which is much smaller than the total number of candidate arcs in a full density 100-node problem (4950). Thus, the algorithm performs quite efficiently, as shown in Table 6.

All the test problems and solutions can be accessed at: http://www.geocities.com/ilhll/mincostflow.html.

Table 6 Results for 100-node non-unit demand

Problem File Name	PS1	PS2	PS3	PS4	PS5	PS6	PS7	PS8	PS9	PS10
Searching Steps	13698	9272	21152	9969	6811	8104	6510	6125	5330	6012
CPU Seconds	18	11	21	10	9	12	7	8	6	7

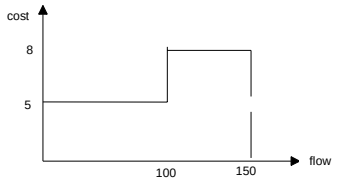


Figure 6-a

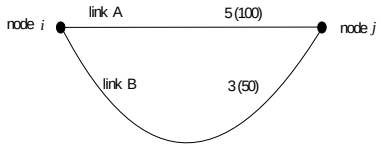


Figure 6-b

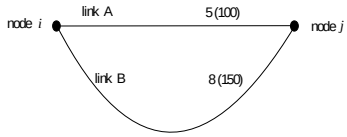


Figure 6-c

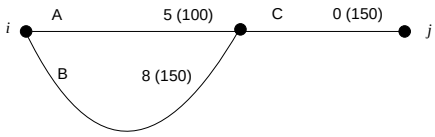


Figure 6-d

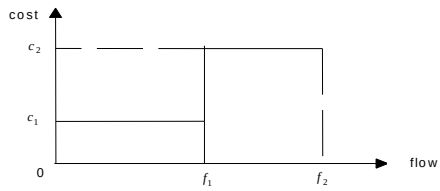


Figure 7-a

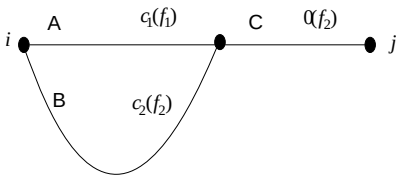


Figure 7-b

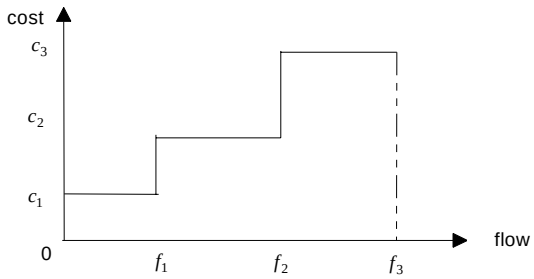


Figure 8-a

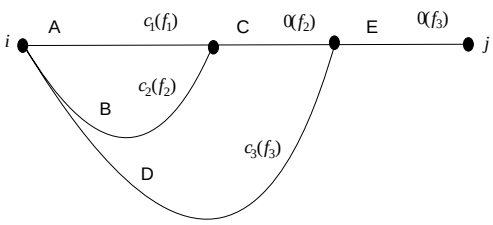


Figure 8-b

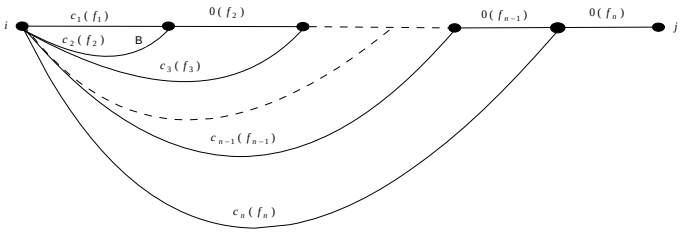


Figure 9

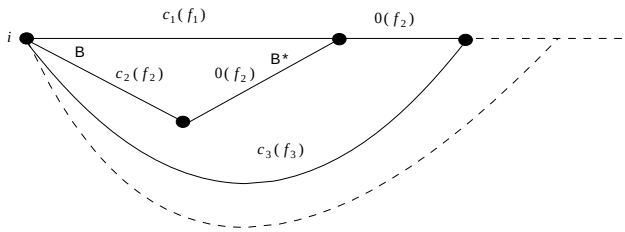


Figure 10

5. Conclusions

We have proposed a class of branch and bound algorithms that can solve up to 100-node minimum cost flow problem (fixed arc costs). Computational experience indicates that the algorithms perform well and demonstrate the efficiency of the bounding method developed. Regarding problems with a non-linear cost function of traffic, we have shown such problems may be modeled in a novel manner using an approach that uses combined constant cost arc to replace a single arc that has a piecewise constant cost function.

References

[1] Ahuja, R., Magnanti, T.; Orlin, J.: *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, New Jersey (1993).

[2] Berry, L.T.M.; Murtagh, B.A., McMahon, G.B., Sugden, S.J. and Welling, L.D. An Integrated GA-LP Approach to Communication Network Design. Telecommunication Systems Journal (1999).

[3] Berry, L.T.M., Murtagh, B.A.; McMahon, G.B. Sugden, S.J. and Welling, L.D., Genetic Algorithms in the Design of Complex Distribution Networks, *International Journal of Physical Distribution and Logistics Management*, 28, No.5, (1998) 377-381.

[4] Minoux, M.: Network Synthesis and Optimal Design Problems: Models, Solution Methods and Applications, *Networks*, 19, (1989) 337-341.

[5] Sharma, U.; Misra, K. and Bhattacharji, A.: Applications of an Efficient Search Technique for Optimal Design of Computer Communication Networks. Micro Electronics and Reliability, 31, (1991) 337 – 341.

[6] Berry, L.T.M., McMahon, G.B., Murtagh, B.A. Sugden, S.J., Optimization Models for Communication Network Design, *Proceedings of the Fourth International Meeting of the Decision Science Institute*, Sydney, July 1997.

[7] Berry, L.T.M., Murtagh, B.A., Sugden, S.J. and McMahon, G.B., Application of a Genetic-based Algorithm for Optimal Design of Tree-structured Communication Networks, P*roceedings of the Regional Teletraffic Engineering Conference of the International Teletraffic Congress*, South Africa, September (1995) 361-370.

[8] Glover, F., Laguna, M., *Tabu Search*, Kluwer Academic Publishers, Boston MA, (1997).

[9] Palmer, C.C., An Approach to a Problem in Network Design Using Genetic Algorithms. *Networks*, 26 (1995) 361-370.

[10] Randall, M., McMahon, G. and Sugden, S., *A Simulated Annealing Approach to Communication Network Design*, TR99-04, School of Information Technology, Bond University, (1999).