

4.1 INTRODUCTION

Trees arise naturally in applications or are used to structure data for fast access. We will study trees in both contexts in this chapter and all the subsequent chapters. This is an indication of the importance of trees.

Definition. A tree $T = (N, A)$ is a connected acyclic graph, where N is a set of *nodes* and A is a set of *arcs* or *edges* that connect pairs of nodes.

Properties. A tree $T = (N, A)$, $|N| = n$, has the following properties :

- n nodes and $n - 1$ arcs.
- No cycles.
- Unique path between any pair of nodes.
- Deleting an arc forms two subtrees.
- Adding an arc forms a cycle.

Spanning Trees. If $G = (N, A)$ is a graph and $T = (N', A')$ is a tree which is a sub-graph of G then T is a *Spanning Tree* of G if $N' = N$, i.e., it is a tree that contains all the nodes of G .

Spanning trees arise frequently in applications. We devote two chapters to *Minimum Spanning Trees* and *Shortest Path Spanning Trees*. Indeed, it can be shown that the feasible and optimal solutions to most network flow problems are spanning trees. The *Network Simplex Method* is the Simplex Method specialized for networks and it pivots not on basic feasible solutions but on basic feasible spanning trees of the underlying network. This is discussed in some detail at the end of this chapter

4.2 DEFINITIONS AND TERMINOLOGY

The following definitions are fairly standard in Computer Science but are not always used in Graph Theory.

1. A tree has a designated node called the *root*.
2. Each node except the root has a *unique parent*.
3. Each node has zero or more *children*.
4. A node with no children is a *leaf node*, otherwise it is called an *internal node*.
5. A *path* between nodes u and v is a sequence of nodes and arcs

$$Path(u, v) = \{u, (u, i), i, (i, j), j, \dots, k, (k, v), v\}.$$

That is, the sequence of nodes and arcs starting at u and ending at v .

6. The *length of a path* is the number of arcs in the path.
7. A node v is an *ancestor* of node u if there path from v to u such that

$$Parent(Parent(\dots Parent(u) \dots)) = v.$$

8. A node v is an *descendant* of node u if there path from u to v such that

$$Parent(Parent(\dots Parent(v) \dots)) = u.$$

9. The *height* of a node u is the number of arcs on the longest path from the u to a leaf node.
10. The *height of a tree* is the height of the root, i.e., the number of arcs on the longest path from *root* to a leaf node.
11. The *depth* of a node is the number of arcs from the node to the root. The depth of the root is 0. ■

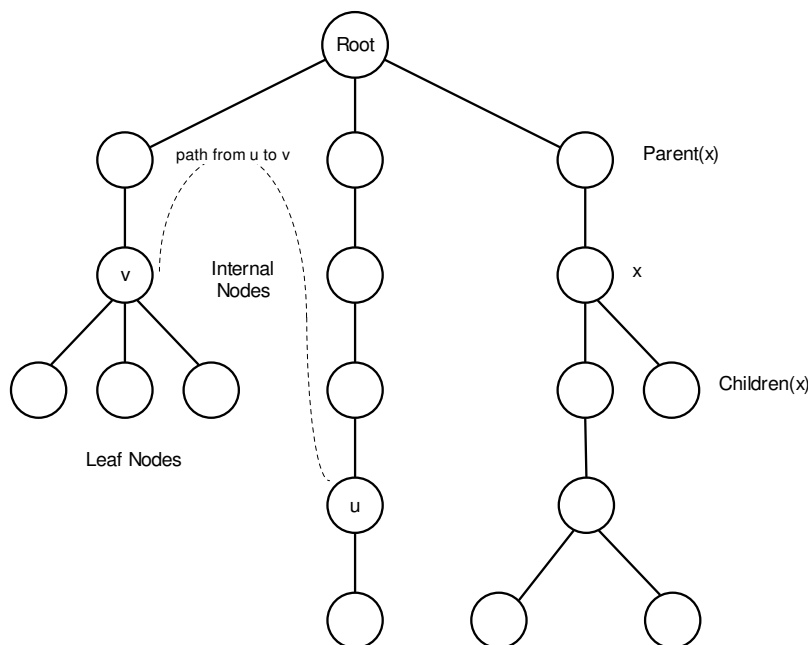


Figure 4.1. Tree terminology : $Height(x) = 3$, $Depth(x) = 2$, $Height(root) = 5$.

A Recursive Definition of Trees. It is very natural to define trees recursively because their structure is recursive. We will see later that many algorithms for tree processing are best expressed recursively. Here is one possible definition :

Definition (Tree).

1. A tree is a single node. This node is the root of the tree,
or
2. A tree is a root node with children c_1, c_2, \dots, c_k and these children are the roots of trees.

■

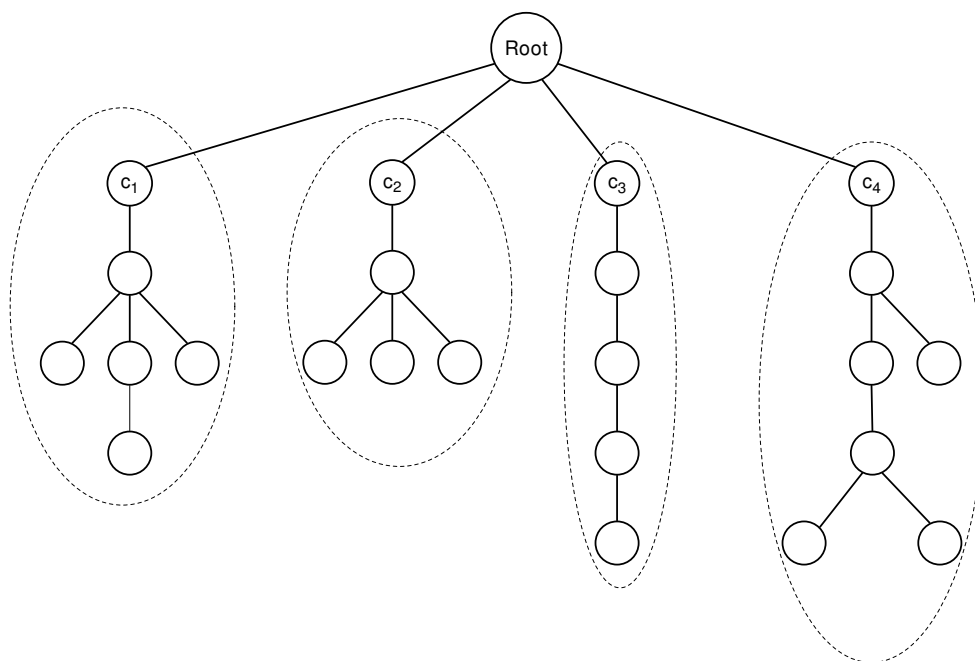


Figure 4.2. A Recursive view of a Tree.

This recursive definition suggests the following recursive tree traversal algorithm which is a prototype for many other tree traversal algorithms.

Tree traversal means ‘visiting’ each node in some order. The order may or may not be important. As each node is visited some processing is done on it which depends on the application. The procedures *PreVisit*, *Visit*, and *PostVisit* perform the application-dependent processing.

Prototype Tree Traversal Algorithm.

A traversal of a tree means ‘visiting’ each node of the tree in some order. The following prototype algorithm uses the recursive definition of a tree to recursively traverse the tree.

<pre> algorithm <i>Traverse</i>(<i>T</i> : <i>node</i>) PreVisit(<i>T</i>) for each <i>c</i> ∈ <i>Child</i>(<i>T</i>) do Traverse(<i>c</i>) endfor PostVisit(<i>T</i>) ENDALG </pre>
--

The procedures *PreVisit*(*T*) and *PostVisit*(*T*) perform some operation on node *u*. *Child*(*T*) is the set of children of node *T*.

4.3 REPRESENTATION OF TREES

We will not define the ADT Tree because the operations would be too numerous and because it is impossible to implement all of them efficiently in one ADT. Instead we will implement specialized tree ADTs as they arise in applications. We will discuss in this chapter the following type of trees :

1. General Trees
2. Binary Trees
3. Binary Search Trees
4. Heaps

We list some general operations and then look at various data structures for representing trees. We analyse each representation to see how much space they require and how efficiently the general operations can be performed on them.

General Operations on Trees.

- Moving up and down paths in a tree.
- Traversing all nodes in a tree.
- Finding, adding, and deleting nodes.
- Finding, adding and deleting arcs.

General Tree Data Structures. There are many ways to represent trees in memory. The most appropriate one depends on the operations we wish to perform. We will use the tree in Figure 4.3 to explain each representation. We label each node with an upper-case letter. We may think of these labels as data contained in the node but their main purpose is to identify each node.

1. **ARRAY OF PARENT POINTERS** : The nodes $i \dots n$ are stored in an array $Node[1..n]$ and the location of the parent of each node i is stored in $Parent[1..n]$. Storage requirement is n cells.
Notice that a node (data) can be stored in any order in the array. With this representation, moving up the tree and adding nodes is easy. All other operations are difficult.
2. **ADJACENCY LIST** : An array $Adj[1..n]$ stores pointers to lists of children, i.e., $Adj[i]$ points to the list of node i 's children. An element on $Adj[i]$ contains the location of a child in array $D[1..n]$.

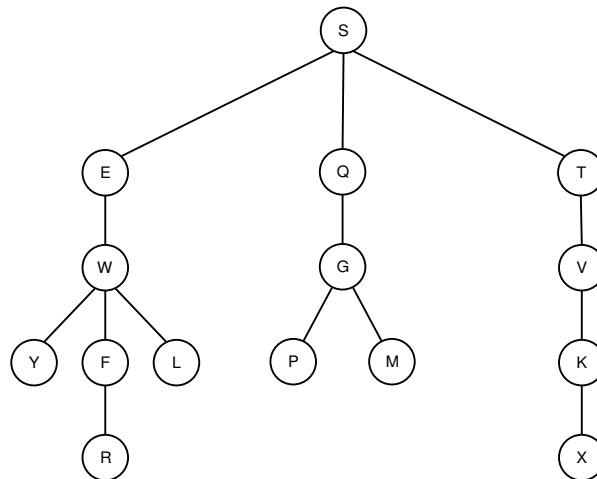


Figure 4.3. A 15-node Tree.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Node	Y	Q	G	S	T	E	K	W	L	X	Y	V	F	M	R
Parent	8	4	2	0	4	4	12	6	8	7	8	13	8	3	13

Figure 4.4. Parent Pointer Representation.

This representation allows easy access down and across the tree. We must add a parent pointer array if we need to move up the tree easily. Storage requirement is $3n$.

3. LEFTMOST CHILD - RIGHT SIBLING : Two arrays $LMC[1..n]$ and $RS[1..n]$ store pointers to the leftmost child and the right sibling, respectively. We add a parent array if we need quick movement up the tree. Storage requirement is $2n$

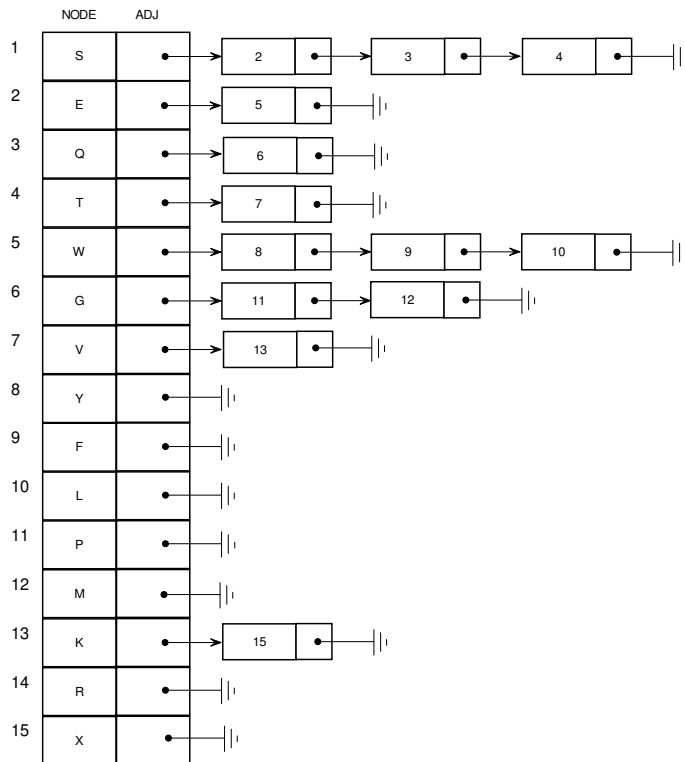


Figure 4.4. Adjacency List representation.

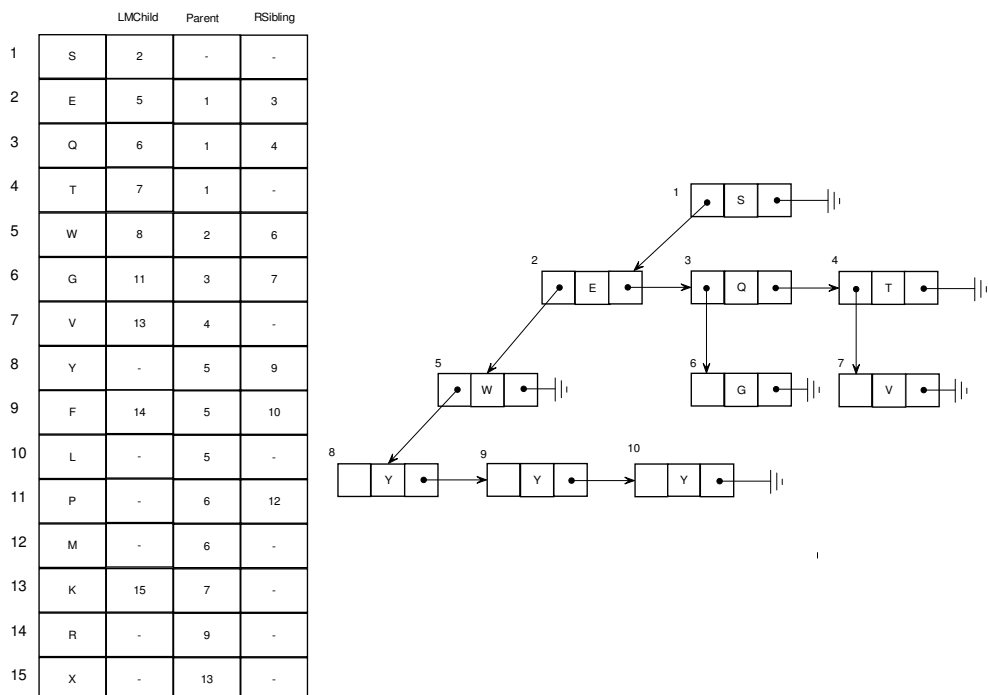


Figure 4.5. Leftmost Child - Right Sibling representation.

4.4 TREE TRAVERSAL ALGORITHMS

Tree traversal algorithms are best described recursively because trees are defined recur-

sively as follows : A Tree is T is a node called the root which has k children C_1, C_2, \dots, C_k that are the roots of trees, called subtrees.

Tree traversal means ‘visiting’ each node in some order. The order may or may not be important. As each node is visited some processing is done on it which depends on the application.

We use modifications of the prototype tree traversal algorithm to get various types of traversals. A traversal that uses PreVisit only is called a *PreOrder* traversal and a traversal that uses PostVisit only is called a *PostOrder* traversal.

Example : . Traverse the tree below in (1) PreOrder and (2) in PostOrder.

PreOrder : S E W Y F R L Q G P M T V K X

PostOrder : Y R F L W E P M G Q X K V T S

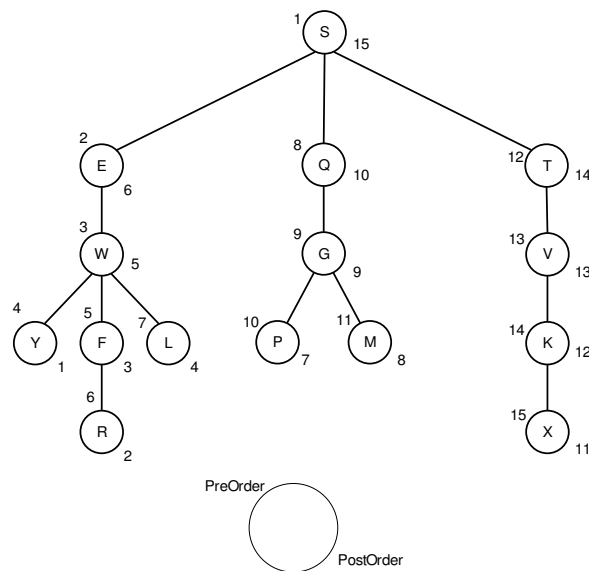


Figure 4.6. Pre- and Post-Order Tree Traversals.

Depth and Breadth First Search. These are two standard traversal algorithms. The first is a particular form of Pre and PostOrder traversal while the second is non-recursive and does not fit the prototype algorithm. More general forms of these algorithms are used to traverse graphs.

<p>algorithm <i>DFS</i> (T : node)</p> <hr/> <pre> PreVisit(T) FOR each child c of T DO DFS (c) ENDFOR PostVisit(T) ENDALG DFS </pre>

```

algorithm BFS ( T : node)
EnQueue(T,Q)
WHILE NOT Empty(Q) DO
  u := DeQueue(Q)
  PreVisit(u)
  FOR each child c of u DO
    EnQueue(c,Q)
  ENDFOR
ENDWHILE
ENDALG BFS

```

Example : Using the 15-node tree above, *DFS* gives the same results as Pre and Post Order traversals. Breadth First Search gives :
 BFS : *S|EQT|WGV|YFLPMK|RX*

Exercise 4.4.1 : Implement an iterative version of DFS. [Hint : Look at BFS and use a stack. Comment]

If each node has a *priority* associated with it then we get *Best First Search* by using a priority queue in BFS. A priority queue keeps the node with the minimum (maximum) priority at the front of the queue. We will discuss priority queues later in this chapter.

Exercise 4.4.1 : *Nearest Common Ancestor.* Write a procedure that efficiently finds the nearest ancestor that is common to two given nodes *u* and *v* in a tree *T*.

4.5 BINARY TREES

Recursive Definition of Binary Trees. A Binary Tree is either

1. empty, or
2. has a root with left and right children which are the roots of binary trees.

Representations for Binary Trees. We can use any general tree representation for Binary Trees but, because each node has at most two children we have special representations for them.

1. **GENERAL BINARY TREE.** Each node is represented by a dynamically-allocated node with the fields *LChild*, *Element*, and *RChild*. The tree is referred to by a pointer to its root.
2. **COMPLETE BINARY TREE.** This is a binary tree in which all levels, except possibly the last, are full. If we number the nodes from top to bottom and left to right, then we can store the tree in a single array without any link information. A node at position *i* has $Parent(i) = i \text{ div } 2$, $LChild(i) = 2 * i$, and $RChild(i) = 2 * i + 1$. The height of a complete binary tree of *n* nodes is $\lfloor \log_2 n \rfloor$. This means that the length of the access path to any node is $O(\log_2 n)$.

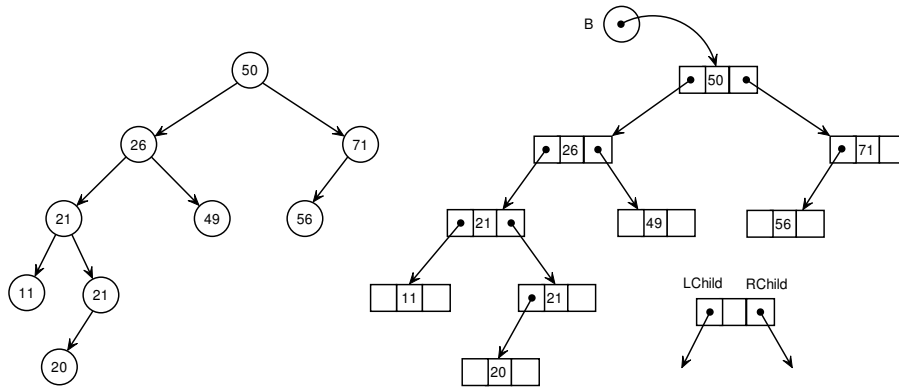


Figure 4.7. Dynamic Binary Tree Representation.

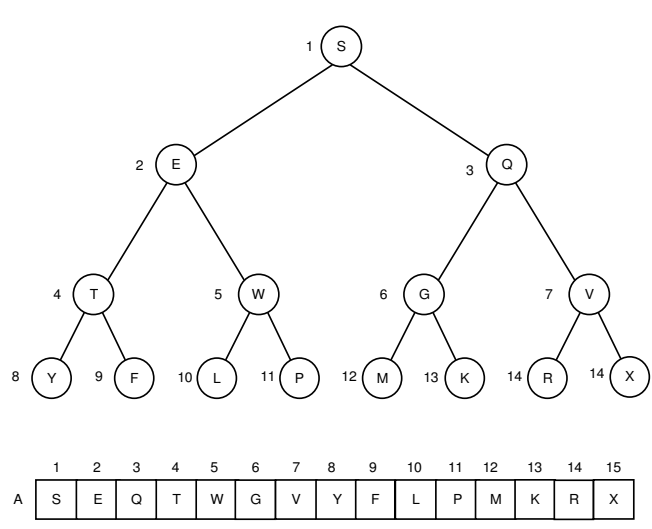


Figure 4.8. Complete Binary Tree representation.

4.5.1 Binary Tree Traversals.

Binary trees, because of their symmetry, allow us to write elegant recursive traversal algorithms. We now define three recursive algorithms for binary trees.

```

algorithm PreOrder (T : node)
    IF NOT Empty(T) THEN
        Visit(T)
        PreOrder(LChild(T))
        PreOrder(RChild(T))
    ENDIF
ENDALG PreOrder
    
```

```

algorithm InOrder (T : node)
    IF NOT Empty(T) THEN
        InOrder(LChild(T))
        Visit(T)
        InOrder(RChild(T))
    ENDIF
ENDALG InOrder
    
```

```

algorithm PostOrder (T : node)
    IF NOT Empty(T) THEN
        PostOrder(LChild(T))
        PostOrder(RChild(T))
        Visit(T)
    ENDIF
ENDALG PostOrder
    
```

Example : Binary Tree Traversals. The binary tree in Figure 8 has the following traversals :

PreOrder: Q W R U O A T E V P S D
 InOrder: U R A O W T Q Y S P D E
 PostOrder: U A O R T W S D P Y E Q

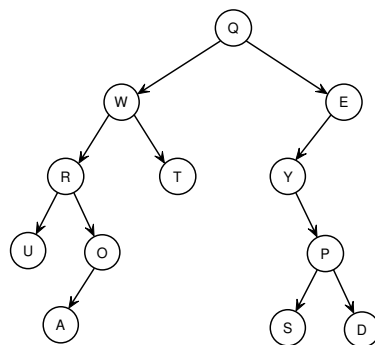


Figure 4.9. Binary Tree Traversals.

Example : (*Expression Trees*). The expression tree for the arithmetic expression

$$A * ((B + C) * (D * E) + F)$$

is shown in Figure 9.

PreOrder: * A + * + B C * D E F
 InOrder: A * B + C * D * E + F
 PostOrder: A B C + D E * * F + * (Reverse Polish)

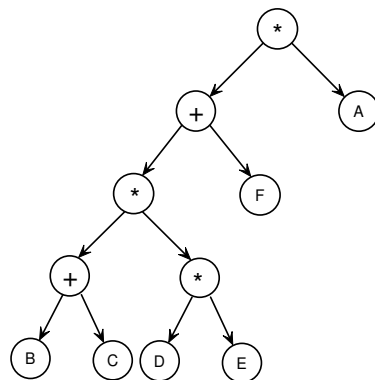


Figure 4.10. Expression Tree.

4.6 BINARY SEARCH TREES

ADT Dictionary. This is a set of elements S totally ordered by some *key* value. The operations are $Member(x, S)$, $Insert(x, S)$ and $Delete(x, S)$.

If we represent the set S as an unordered array then $Insert$ is $O(1)$ and $Delete$ and $Member$ are $O(n)$. If the array is sorted then these become $O(n)$, $O(1)$ and $O(\log_2 n)$, respectively, assuming we use *Binary Search* for $Member$.

There are two common methods for efficiently implementing the ADT Dictionary :

1. *Binary Search Tree* and
2. *Hash Table*.

Binary Search Trees are binary trees which are ordered as follows :

$$Key(LChild(i)) < Key(i), \text{ and } Key(RChild(i)) > Key(i).$$

Binary Search Tree Properties. The ordering above implies the following properties :

- Each subtree is a binary search tree.
- The key values of the left subtree of a node are all less than the key value of the node.
- The key values of the right subtree of a node are all greater than the key value of the node.
- The minimum (maximum) key value is in the left (right) - most node.
- The height of a binary search tree is in the range $[\log_2 n \dots n - 1]$.

Example : $S = \{50, 26, 49, 21, 71, 21, 20, 48, 11, 56\}$..

Using the three-field dynamic node representation the three dictionary operations can be implemented recursively as follows :

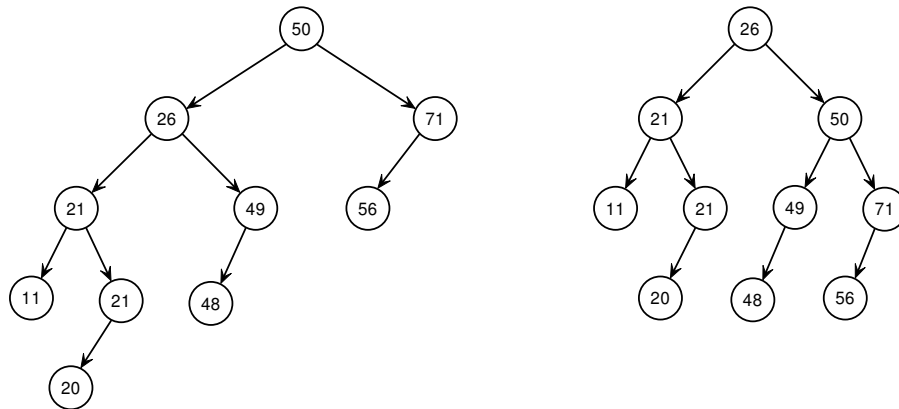


Figure 4.11. Two Binary Search Trees for the same set S .

```

function Member (x:elemtype, S:BSTreeType): BOOLEAN


---


if S = NIL then
    Member := FALSE
else x = S^.elem then
    Member := TRUE
else x < S^.elem then
    Member := Member(x, S^.LChild)
else x > S^.elem then
    Member := Member(x, S^.RChild)
end;
end;{ FUNC Member }
    
```

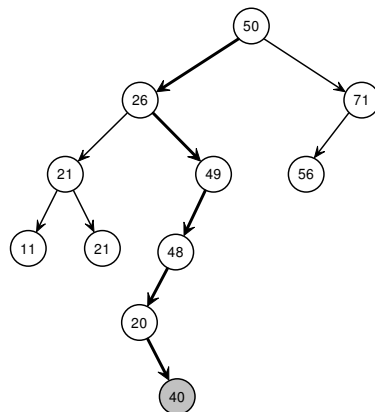


Figure 4.11. Binary Search Tree Insertion.

```

procedure Insert (x, S);


---


  if S = NIL then begin
    New(S);
    S^.elem := x;
    S^.LChild := NIL;
    S^.RChild := NIL;
  end
  else if x < S^.elem then
    Insert(x, S^.LChild)
  else if x > S^.elem then
    Insert(x, S^.RChild)
  end;
end;{ PROC Insert }

```

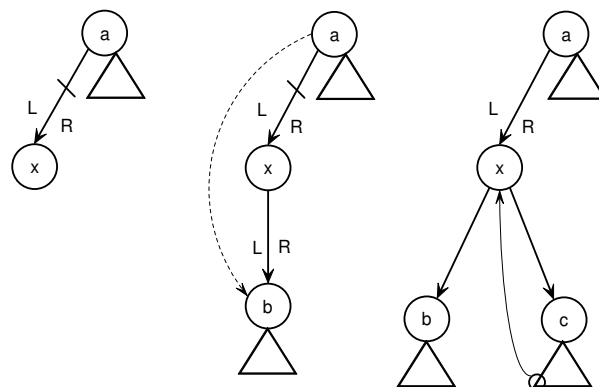


Figure 4.12. The Delete Operation.

```

procedure Delete (x:elemtype,S:BSTreeType);


---


  if S < NIL then
    if x < S^.elem then
      Delete(x, S^.LChild)
    else if x > S^.elem then
      Delete(x, S^.RChild)
    else if S^.LChild = NIL then
      S := S^.RChild
    else if S^.RChild = NIL then
      S := S^.LChild
    else
      S^.elem := DeleteMin(S^.RChild)
    end;
  end;
end;{ PROC Delete }

```

The *DeleteMin* operation is easy : by the binary search tree property the minimum (maximum) is the left-(right-)most node. To find the minimum we start at the root and go left until we find a node with no left child. This is the minimum node. The algorithm is as follows:

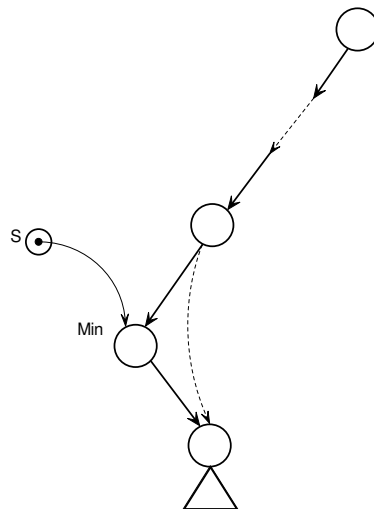


Figure 4.13. The *DeleteMin* Operation.

```

function DeleteMin (S:BSTreeType):elemtype;


---


    if S^.LChild = NIL then begin
        DeleteMin := S^.elem
        S := S^.RChild;
    end
    else DeleteMin := DeleteMin(S^.LChild);
end;{ FUNC DeleteMin }
    
```

Analysis of Binary Search Trees. It can be shown that the average height of a randomly generated binary search tree is $O(\log_2 n)$. This means that all operations on such trees are $O(\log_2 n)$, on average.

TO BE EXPANDED LATER

4.7 ADT PRIORITY QUEUE

This is a very useful ADT that is used in many applications, especially in Operations Research. It is useful any time we to repeatedly find the minimum or maximum of a totally-ordered set that is constantly changing (it key values).

Definition (*Priority Queue*).

- Set and Structure : A set of elements S totally ordered by some *key* value.
- Operations :
 1. $Insert(x, S') \rightarrow S'' = S' \cup \{x\}$
 2. $DeleteMin(S') \rightarrow x$ with min key value $\in S', S'' = S' - \{x\}$.
- Representations : Array, list, or partially ordered complete binary tree. This is called a *Heap*.



4.7.1 Heap Representation.

If we represent the set S as an unordered array or list then $Insert$ is $O(1)$ and $DeleteMin$ is $O(n)$. If the array or list is sorted then these become $O(n)$ and $O(1)$, respectively.

We can do much better if we represent S as a *partially ordered tree* or *Heap* . This is a tree in which *the key value of any node is less than or equal to those of its children*. This implies the *Partially Ordered Tree Property* : the minimum of any subtree is at the root of the subtree. This means that the minimum of S is at the root of the partially ordered tree representing S .

The most efficient way to represent a partially ordered tree is a *Heap*. A heap is a complete binary tree arranged so that

$$key(i) \leq key(LChild(i)) \text{ and } key(RChild(i)).$$

For example if we represent $S = \{50, 26, 49, 21, 71, 21, 20, 48, 11, 56\}$ as a heap we get the tree shown in Figure 9. Because a heap is a complete binary tree it height is $O(\log_2 n)$.

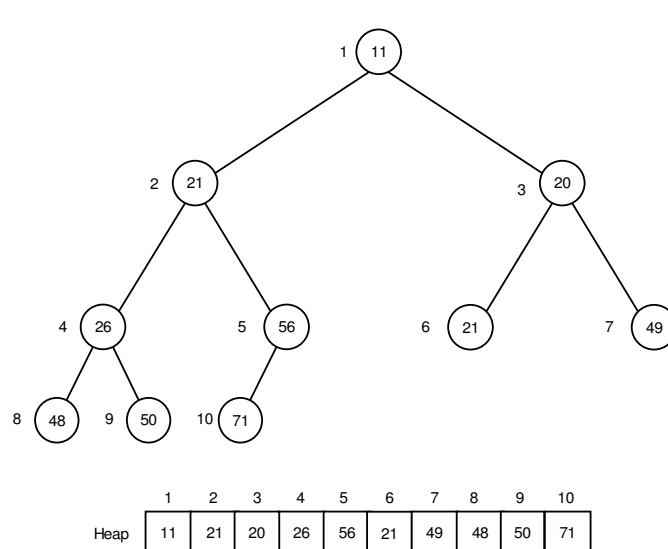


Figure 4.14. A Min 2-Heap.

SiftUp and SiftDown Operations.

Before we implement the *Insert* and *DeleteMin* operations we need two ancilliary operations : *SiftUp*(i, S) moves the contents of node i up to its correct position in the heap and *SiftDown*($i, S, size$) moves the contents of node i down to its correct position.

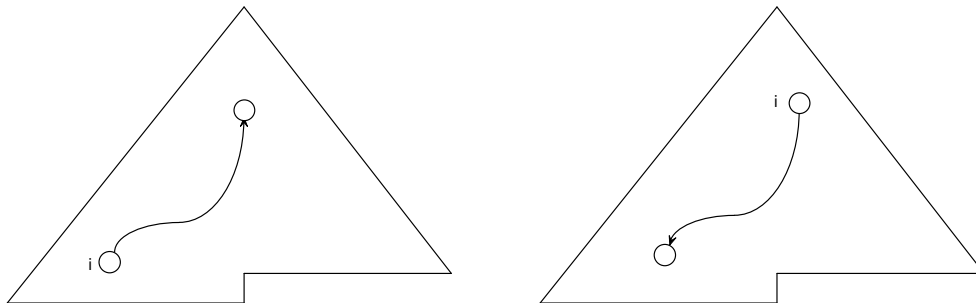


Figure 4.15. SiftUp and SiftDown Operations.

```

procedure SiftUp( $i, \text{Heap}, \text{HSize}$ );
{ Moves heapnode  $i$  in position  $\text{Heap}[i]$  up to its correct position in  $\text{Heap}$  }
   $p := i \text{ DIV } 2$ 
  IF  $\text{Heap}[p] \geq \text{Heap}[i]$  THEN
    Swap( $\text{Heap}[i], \text{Heap}[p]$ )
    SiftUp( $p, \text{Heap}$ )
  ENDIF;
END; { ALG SiftUp }

```

```

procedure SiftDown( $i, \text{Heap}, \text{HSize}$ )
{ Moves heapnode  $i$  in position  $\text{Heap}[i]$  down to its correct position in  $\text{Heap}$  }
   $c := 2*i$ 
  IF  $c > \text{HSize}$  THEN
    IF  $\text{Heap}[c+1] < \text{Heap}[c]$  THEN  $\text{Inc}(c)$  ENDIF
  ENDIF
  IF  $c = \text{HSize}$  AND ( $\text{Heap}[i] < \text{Heap}[c]$ ) THEN
    Swap( $\text{Heap}[i], \text{Heap}[c]$ )
    SiftDown( $c, \text{Heap}, \text{HSize}$ )
  ENDIF
END { PROC SiftDown }

```

These operations affect only one path in the heap. Hence they require $O(\log_2 n)$ time.

Exercise 4.7.1 : Write iterative versions of *SiftUp* and *SiftDown*.

4.7.2 Insert and DeleteMin Operations.

We are now in a position to implement the *Insert* and *DeleteMin* operations.

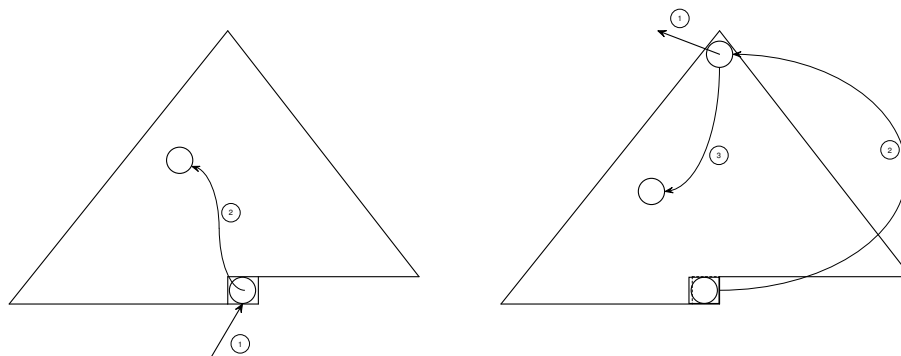


Figure 4.16. The *Insert* and *DeleteMin* Operations.

Insert(x, S) is easily implemented by inserting x in the next available space in the heap array S . *SiftUp* then moves x up to its correct position. The algorithm is as follows :

<pre> algorithm <i>Insert</i> ($x, \text{Heap}, \text{Hsize}$) Inc(Hsize) Heap[Hsize] := x SiftUp(Hsize, Heap, Hsize) endalg <i>Insert</i> </pre>

DeleteMin(S) is slightly more difficult

<pre> algorithm <i>DeleteMin</i>($\text{Heap}, \text{Hsize}$) : <i>element</i> DeleteMin := Heap[1] Heap[1] := Heap[size] Dec(Hsize) SiftDown(1, Heap, Hsize) endalg <i>DeleteMin</i> </pre>

It is obvious that both of these operations are $O(\log_2 n)$.

We may create the initial heap in two ways. The first is straight-forward : Read in each x and use *Insert*($x, S, size$), incrementing $size$ for each x read in. This method is $O(n \log_2 n)$. A more elegant and efficient method is to assume that all elements are in the array S , in any order, and then apply the following algorithm :

```

algorithm MakeHeap(S, size)


---


  for i := n DIV 2 downto 1 do
    SiftDown(i,S,size)
  endfor
endalg MakeHeap
    
```

This algorithm requires $n/2$ SiftDown operations and so the complexity is, on the face of it, $O(n \log_2 n)$. Surprisingly, a more careful analysis shows that this algorithm is $O(n)$. We now do this analysis.

Analysis of MakeHeap. We wish to find a formula for $T_{mh}(n)$, the number of exchanges required to make a heap of n nodes. Assume, for convenience, that all levels of the heap are full. There is 1 node at level 0, 2 at level 1, 4 at level 2, and 2^i at level i . Hence, for a heap with n nodes and l levels we have the formula

$$n = \sum_{i=0}^l 2^i = 2^{l+1} - 1.$$

Thus we see that a heap with n nodes has

$$l = \log_2(n + 1) - 1.$$

Now, in MakeHeap, SiftDown operates at all levels except the lowest, l , exchanging node values until each finds its proper place. This means that

- nodes at level l require 0 exchanges
- nodes at level $l - 1$ require 1 exchange
- nodes at level $l - 2$ require 2 exchanges
- nodes at level $l - i$ require i exchanges
- nodes at level 0 require l exchanges

There are 2^{l-i} nodes at level $l - i$ and each of these requires i interchanges, at most. Thus there are, at most, $i2^{l-i}$ interchanges at level i . Hence, the total number of interchanges is

$$T_{mh}(n) = \sum_{i=0}^l i2^{l-i} = \sum_{i=0}^l i2^l 2^{-i} = 2^l \sum_{i=0}^l i2^{-i}.$$

Consider the $\sum_{i=0}^l i2^{-i}$ part of the formula above. This is bounded above by $\sum_{i=0}^{\infty} i2^{-i}$. This is a standard summation whose value is obtained as follows :

$$\begin{aligned}
 S &= \sum_{i=0}^{\infty} i \frac{1}{2^i} = \sum_{i=0}^{\infty} (i + 1) \frac{1}{2^{i+1}} \\
 &= \frac{1}{2} \sum_{i=0}^{\infty} i \frac{1}{2^i} + \frac{1}{2} \sum_{i=0}^{\infty} \frac{1}{2^i} \\
 &= \frac{1}{2} S + \frac{1}{2} \left(\frac{1}{1 - 1/2} \right) \\
 &= \frac{1}{2} S + 1.
 \end{aligned}$$

Thus

$$S = \sum_{i=0}^{\infty} i2^{-i} = 2.$$

We now plug this into the main formula to get the following upper bound :

$$T_{mh}(n) = 2^l \sum_{i=0}^l i2^{-i} < 2^l S = 2^{l+1} = n + 1.$$

Thus we have shown that MakeHeap can be performed in (usually less than) n exchanges.

4.8 APPLICATION OF HEAPS TO SORTING

We wish to arrange the n numbers of a table X in ascending order. The following algorithm, called *Selection Sort*, is one of the simplest.

<pre> algorithm <i>SelectSort</i> (X, n) <hr/> for $i := n$ downto 2 do $i^* := \text{LocateMax}(X, i)$ $\text{Swap}(X[i], X[i^*])$ endfor endalg <i>SelectSort</i> </pre>
--

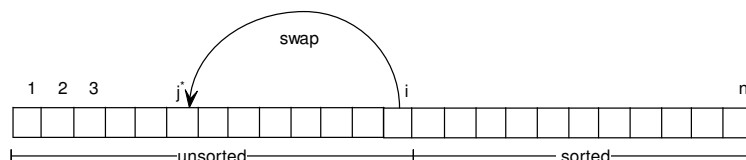


Figure 4.17. Selection Sort.

The operation of this algorithm is shown in Figure 4.17. The function $\text{LocateMax}(X, i)$ finds the location of the maximum element in the sub-table $X[1] \dots X[i]$. This requires $i - 1$ comparisons. The Swap operation is performed in constant time. The LocateMax function and the Swap operation are used $n - 1$ times. Hence, the total number of operations performed is

$$T_s(n) = \sum_{i=1}^{n-1} [(i - 1) + 1] = O(n^2).$$

We now use a heap and its operations to speed up the SelectSort algorithm. In that algorithm we repeatedly swapped the maximum with the last element of the unsorted sub-table $X[1] \dots X[i]$. We initially form the array X into a heap and this puts the maximum in position 1 of the array. Put the maximum in its final position by swapping the values in positions 1 and n . Ignore position n from now on and consider the heap defined by $X[1] \dots X[n - 1]$. Because of the swap operation the element at the top of this heap is

probably not in its correct position. Sift the top element down to its correct position. We now have the second-largest element of X at the top of the heap and we can repeat the steps above until we have a heap of size 1. These steps are succinctly defined by the following algorithm.

```
algorithm HeapSort ( $X, n$ )
  for  $i := n \text{ DIV } 2$  downto 1 do
    SiftDown( $i, X, n$ )
  endfor
  for  $i := n$  downto 2 do
    Swap( $X[1], X[i]$ )
    SiftDown(1,  $X, i - 1$ )
  endfor
endalg HeapSort
```

Analysis of HeapSort. The first for - loop forms the initial heap. We have shown that this requires $O(n)$ time. The second for - loop repeatedly swaps the maximum $X[1]$ with the last element in $X[1] \dots X[i]$ and re-forms the heap with the SiftDown operation. This is the same as SelectSort with SiftDown replacing LocateMax. The second loop requires $n - 1$ SiftDown operations. Thus we get an $O(n \log_2 n)$ sorting algorithm.

SelectSort takes 4 secs for $n = 500$ and 16 secs for $n = 1000$ on a 10MHz AT computer. *HeapSort* takes 1.15 sec for $n = 1,000$ and 12.87 secs for $n = 30,000$ on a 10MHz AT. This algorithm is robust, i.e., it is guaranteed to give $O(n \log_2 n)$ time for all inputs because the height of the tree is $O(\log_2 n)$ by construction.

We have seen how a good data structure – a heap – can be used to significantly speed up an elementary algorithm – SelectSort. In this case the algorithm is within a multiplicative constant of the optimum.

4.8.1 d -Heaps.

These are the same as 2-heaps except each node has d children. This means that the height of a d -heap is $O(\log_d n)$.

A node at position i has

$$\text{Parent}(i) = \lceil (i - 1)/d \rceil,$$

$$\text{Children}(i) = [d(i - 1) + 2 \dots \min\{di + 1, n\}].$$

The *SiftUp* operation is $O(\log_d n)$ because this is the height of the tree. The *SiftDown* operation is a little trickier because we have to choose the minimum of d possible children at each step. This means that the complexity of the *SiftDown* operation is $O(d \log_d n)$.

What value of d is best? This depends on the application. For example, if there are many *SiftUp* and few *SiftDown* operations then large value of d is better than a small value. In general, if an application requires k_1 *SiftUps* and k_2 *SiftDowns*, the the total complexity is

$$T(n, d) = k_1 \log_d n + k_2 d \log_d n.$$

This has a minimum when $d = \lceil 2 + k_1/k_2 \rceil$.

Thus, if we are using a heap in an application, and we know it requires roughly k_1 *SiftUps* and k_2 *SiftDowns* then can *tune* the heap for best performance. Even if we do not know k_1 and k_2 we can do trial runs to estimate these parameters and then tune the heap. We will see an application of these ideas when we study shortest path algorithms.

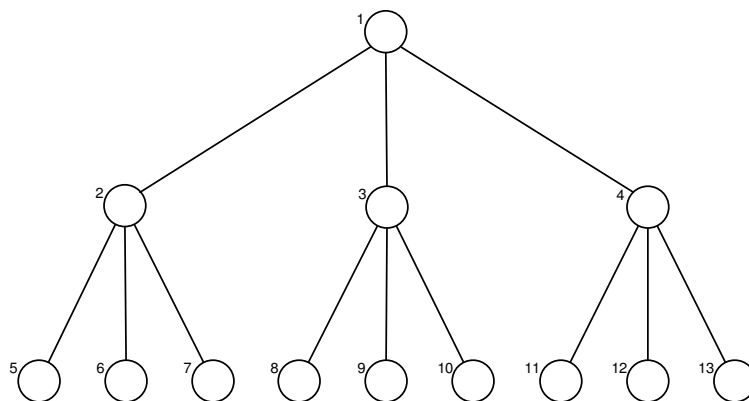


Figure 4.18. A 3-Heap.

4.9 APPLICATION OF TREES TO THE NETWORK SIMPLEX METHOD

The following is a paper written in 1998 based in part on a seminar given to the *Operations Research Society of Ireland* in April 1982 and in part on the 1991 M.Mangt.Sc. dissertation by O'Neill and Kelly.

4.10 HISTORICAL REVIEW

The transportation problem is a special case of the linear programming problem and it was first studied by the Russian mathematician, L.V. Kantorovich, in a paper entitled *Mathematical Methods of Organizing and Planning Production* (1939). Largely ignored by his Russian contemporaries, his work remained unknown elsewhere until 1960, long after the seminal Western work in this field had been completed. However, its value was recognised in 1975 by the Nobel Academy, which awarded its Prize in Economic Science in equal shares to Kantorovich and T.C. Koopmans 'for their contribution to the theory of optimal allocation of resources'.

Kantorovich (1939) deals with nine different problem types, with the stated goal of aiding the third of Stalin's five-year plans to obtain the greatest possible usage of existing industrial resources. Principal among these problems were:

- the distribution of work among individual machines
- the distribution of orders among enterprises
- the distribution of raw materials, fuel and factors of production
- the minimisation of cutting scrap
- the best plan of freight shipments

A solution method was also included and was described by Marchenko, in a forward to the article, as "an original method going beyond the limits of classical mathematical analysis". In retrospect, the algorithm is rudimentary and incomplete, but its importance lies in the application of the method to "questions of organizing production". A more complete theory of the transshipment problem is to be found in *On the Translocation of Masses*, Kantorovich in 1942 and in Kantorovich and Gauvrin in 1949. The relations between primal and dual are recognized and an extension to capacitated networks is given.

In the West, the seminal work in the area was done by F. Hitchcock (1941) who outlines the standard form of the transportation problem for the first time. He proposes a $m \times n$ dimensional geometric interpretation of the transportation of goods from m factories to n cities, and constructs a "region of possibilities" on whose boundary the optimal solution must lie. A method for finding the fixed points on this boundary, called *vertices*, is proposed, and is shown to iteratively generate better solutions by expressing the objective function in terms of variables with zero values. Hitchcock also notes the occurrence of multiple optimal solutions and degeneracy. At around the same time, T.C. Koopmans, as a member of the Combined Shipping Board during the Second World War, began research into reducing shipping times for the transportation of cargo. In *Optimum Utilisation of the Transportation System* in 1947, he studies the transportation problem, developing the idea of node-potentials and an optimality criterion. He also shows that an extreme point, or vertex as Hitchcock terms it, is represented by a tree. It is because of the work of these two researchers that the classical transportation problem is often referred to as the *Hitchcock-Koopmans Transportation Problem*.

The Linear Programming Problem and the Simplex Algorithm for its solution were formally stated by G.B Dantzig in 1947. The first instance of a Linear Programming Problem is attributed to the nineteenth century mathematician Fourier, who in 1826 studied linear inequalities to find the "least maximum deviation" fit to a system of linear equations. He reduced this problem to that of finding the optimal vertex in a solution space and suggested a method of iterative, vertex to vertex, improvement. This principle is the basis of the simplex algorithm as we know it.

The immediate predecessor of the Simplex Method was the *Input-Output* model of the national economy proposed by W. Leontief in 1936. This quantitative model was designed to trace and predict the effect of government policy and consumer trends on the various sectors of the economy, all of which were interlinked.

The accepted father of linear programming and the Simplex Method is G.B. Dantzig. From 1946 to 1947, Dantzig worked to mechanise the planning processes of the United States Air Force. He generalized the Input-Output model and, finding no method of solving the problem extant in the literature, he designed the Simplex Method. The Simplex Method could be applied to transportation problems, and went on to develop a specialisation of his method for these problems. Having expressed the problem in terms of a source-destination tableau, he describes the calculation of simplex multipliers and reduced costs to iterate towards the optimal solution. The method is known as *MODI* - “modified for distribution”. Dantzig noted at this point the triangularity of the basis matrix and integrality of solutions generated from problems with integer parameters. He also examines the possibilities of degeneracy and cycling, and proposes a perturbation rule to avoid them. Alex Orden (1956) generalised Dantzig’s method and applied it to the *transshipment problem*, allowing the inclusion of nodes with zero supply and demand. This is achieved by representing every node as both a source of supply and a demand destination, and adding a buffer amount, greater than or equal to the total supply, to every node. The revised formulation is solved using Dantzig’s transportation method with flows from a node to itself being ignored.

In 1955, Dantzig developed a *bounded variable* simplex method for linear programming. This method allowed the handling of upper bounds on variables without the inclusion of explicit constraints. It was originally envisaged for the problem of scheduling requests for computing time with a restriction on the number of available computer-hours per day. The method operates in a similar fashion to the transportation method, but variables can leave when they reach either their upper bounds or lower bounds (generally zero). In 1962, Dantzig formulated this method in general terms for the solution of a bounded transportation problem. Dantzig (1963) contains comprehensive coverage of all the above developments.

The transportation problem is a special case of the Minimum Cost Flow (MCF) problem which was first thoroughly studied by Ford and Fulkerson (1962). This monograph contains an extensive review and analysis of the MCF problem and its specialisations, and also includes an innovative primal-dual solution algorithm. These developments led directly to the development of the Network Simplex Method (NSM) as a primal solution method for Minimum Cost Flow Problems, which may be described as bounded transshipment problems. NSM is essentially the graph-theoretical (as opposed to linear-algebraic) representation of Dantzig’s bounded simplex method.

For a complete and up-to-date account of network flows and the MCF problem see the book by Ahuja, Magnanti, and Orlin (1993).

4.11 FORMULATION AND SPECIALISATIONS

Let $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ be a directed network consisting of a finite set of nodes, \mathcal{N} , and a set of directed arcs, \mathcal{A} linking pairs of nodes in \mathcal{N} . We associate with every arc of $(i, j) \in \mathcal{A}$, a flow x_{ij} , a cost per unit flow c_{ij} , a lower bound on the flow l_{ij} and an upper bound or capacity u_{ij} .

To each node $i \in \mathcal{N}$ we assign an integer number b_i representing the available supply of, or demand for flow at that node. If $b_i > 0$ then node i is a supply node, if $b_i < 0$ then node i is a demand node, and otherwise, where $b_i = 0$, node i is referred to as a transshipment node. Total supply must equal total demand or total net supply must be zero, i.e., $\sum b_i = 0, i \in \mathcal{N}$.

The formulation of the problem as a Linear Programming problem is as follows:

$$\text{minimise } \sum_{(i,j) \in \mathcal{A}} c_{ij}x_{ij} \tag{1.1}$$

subject to

$$\sum_{j:(i,j) \in \mathcal{A}} x_{ij} - \sum_{j:(j,i) \in \mathcal{A}} x_{ji} = b_i, \quad \text{for all } i \in \mathcal{N} \tag{1.2}$$

$$0 \leq l_{ij} \leq x_{ij} \leq u_{ij}, \quad \text{for all } (i, j) \in \mathcal{A} \tag{1.3}$$

The Minimum Cost Flow (MCF) Problem is to send the required flows from the supply nodes to the demand nodes (i.e. satisfying the demand constraints (1.2), at minimum cost (1.1). The *flow bound* constraints, (1.3), must be satisfied. The demand constraints are also known as *mass balance* or *flow balance* constraints. The total net supply must equal zero and summing the flow balance equations over all $i \in \mathcal{N}$ we get

$$\sum_{i \in \mathcal{N}} \left(\sum_{j: (i,j) \in \mathcal{A}} x_{ij} - \sum_{j: (j,i) \in \mathcal{A}} x_{ji} \right) = \sum_{i \in \mathcal{N}} b_i = 0.$$

This means that constraints (1.2) are linearly dependent and so we may arbitrarily drop one of these. The problem in matrix notation is

$$\text{minimize } \{cx \mid Nx = b \text{ and } 0 \leq l \leq x \leq u\}$$

where N is a node-arc incidence matrix having a row for each node and a column for each arc. The MCF formulation is particularly broad and can be used as a template upon which a number of network problems may be modelled. The following are some examples.

4.11.1 The Shortest Paths Problem.

The shortest path problem is to find the directed paths of shortest length from a given *root* node to all other nodes. We assume, without loss of generality that the root node is 1, and set the parameters of MCF as follows: $b(1) = n - 1$, $b_i = -1$ for all other nodes, c_{ij} is the length of arc (i, j) , $l_{ij} = 0$ and $u_{ij} = n - 1$ for all arcs. The LP formulation is:

$$\text{minimise } \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij}$$

subject to

$$\begin{aligned} \sum_{j: (1,j) \in \mathcal{A}} x_{1j} - \sum_{j: (j,1) \in \mathcal{A}} x_{j1} &= n - 1 \\ \sum_{j: (i,j) \in \mathcal{A}} x_{ij} - \sum_{j: (j,i) \in \mathcal{A}} x_{ji} &= -1, \quad \text{for } i = 2, 3, \dots, n \\ 0 \leq x_{ij} &\leq n - 1, \quad \text{for all } (i, j) \in \mathcal{A} \end{aligned}$$

The optimum solution to this problem sends unit flow from the root to every other node along a shortest path.

4.11.2 The Maximum Flow Problem.

The maximum flow problem is to send the maximum possible flow from a specified *source* node (node 1) to a specified *sink* node (node n). The arc $(n, 1)$ is added with $c_{n1} = -1$, $l_{n1} = 0$ and $u_{n1} = \infty$. The supply at each node is set to zero, i.e. $b_i = 0$ for all $i \in \mathcal{N}$, as is the cost of each arc $(i, j) \in \mathcal{A}$. Finally, u_{ij} represents the upper bound on flow in arc (i, j) to give the following LP formulation:

$$\text{minimise } -x_{n1}$$

subject to

$$\begin{aligned} \sum_{j: (i,j) \in \mathcal{A}} x_{ij} - \sum_{j: (j,i) \in \mathcal{A}} x_{ji} &= 0, \quad \text{for all } i \in \mathcal{N} \\ 0 \leq x_{ij} &\leq u_{ij}, \quad \text{for all } (i, j) \in \mathcal{A} \\ 0 \leq x_{n1} &\leq \infty \end{aligned}$$

4.11.3 The Transshipment Problem.

The Transshipment problem is an MCF problem with transshipment nodes and unlimited arc capacity. We divide the nodes into three subsets: \mathcal{N}_1 , a set of supply nodes, \mathcal{N}_2 , a set of demand nodes and \mathcal{N}_3 , a set of transshipment nodes.

The LP formulation is:

$$\text{minimise } \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij}$$

subject to

$$\sum_{j:(i,j) \in \mathcal{A}} x_{ij} - \sum_{j:(j,i) \in \mathcal{A}} x_{ji} = b_i, \quad \text{for all } i \in \mathcal{N}_1, \mathcal{N}_2$$

$$\sum_{i:(i,j) \in \mathcal{A}} x_{ij} - \sum_{j:(j,i) \in \mathcal{A}} x_{ji} = 0, \quad \text{for all } j \in \mathcal{N}_3$$

$$x_{ij} \geq 0, \quad \text{for all } (i,j) \in \mathcal{A}$$

4.11.4 The Transportation Problem.

a special case of the transshipment problem where there are no transshipment nodes, i.e., the nodes are either supply or demand nodes. This means that the underlying graph is bi-partite, where \mathcal{N}_1 is the set of *source* nodes, and \mathcal{N}_2 is the set of sink or *destination* nodes, such that $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$, and the set of arcs is defined by $\mathcal{A} = \{(i, j) | i \in \mathcal{N}_1, j \in \mathcal{N}_2\}$

The objective is to find the least cost shipping plan from the sources of supply (\mathcal{N}_1) to the destinations (\mathcal{N}_2), where x_{ij} is the number of units shipped from source i to destination j , and c_{ij} is the cost of shipping one unit from i to j . The supply and demands at sources and destinations respectively are denoted by b_i . There are no upper bounds on flows. (A problem with no upper flow bounds is said to be *uncapacitated*).

In LP form, the problem is stated:

$$\text{minimise } \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij}$$

subject to

$$\begin{aligned} \sum_{j:(i,j) \in \mathcal{A}} x_{ij} &= b_i, & \text{for all } i \in \mathcal{N}_1 \\ \sum_{i:(i,j) \in \mathcal{A}} x_{ij} &= -b_j, & \text{for all } j \in \mathcal{N}_2 \\ x_{ij} &\geq 0, & \text{for all } (i, j) \in \mathcal{A} \end{aligned}$$

4.11.5 The Assignment Problem.

This is a special case of the transportation problem in which each supply is 1 and each demand is 1, and $m = n$. The graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ is composed as follows: $\mathcal{N} = \mathcal{N}_1 \cup \mathcal{N}_2$ where \mathcal{N}_1 and \mathcal{N}_2 are two disjoint sets of equal size. The problem is to assign each element in \mathcal{N}_1 to one element in \mathcal{N}_2 (e.g. people to machines), at minimum cost. If element $i \in \mathcal{N}_1$ is assigned to element $j \in \mathcal{N}_2$, then $x_{ij} = 1$. Otherwise $x_{ij} = 0$. The cost of assigning $i \in \mathcal{N}_1$ to $j \in \mathcal{N}_2$ is represented by c_{ij} . For each $i \in \mathcal{N}_1, b_i = 1$, and for each $j \in \mathcal{N}_2, b_j = -1$. All lower bounds are 0, and all upper bounds are 1.

The LP formulation is:

$$\text{minimise } \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij}$$

subject to

$$\begin{aligned} \sum_{j:(i,j) \in \mathcal{A}} x_{ij} &= 1, & \text{for all } i \in \mathcal{N}_1 \\ \sum_{i:(i,j) \in \mathcal{A}} x_{ij} &= -1, & \text{for all } j \in \mathcal{N}_2 \\ x_{ij} &\in \{0, 1\}, & \text{for all } (i, j) \in \mathcal{A} \end{aligned}$$

4.12 SOLUTION METHODS

A variety of solution methods for the MCF problem are described in the literature. These may be categorised into a number of principal approaches, namely *primal*, *dual*, *primal-dual* and *scaling* algorithms. For example, *negative cycle* and *primal simplex* are both primal algorithms, while dual methods include *successive shortest paths* and a specialisation of the *dual simplex* method for LP problems. Ford's *Out-of-Kilter* algorithm is perhaps the best known primal-dual method. Finally, there exist a new family of relaxation algorithms based on either *right-hand-side* or *cost scaling*.

The *Network Simplex Method* is a specialization of the bounded variable primal simplex algorithm, specifically for the MCF problem. The key idea in this method is that any basic feasible solution of the MCF is a spanning tree of the underlying graph. Hence for each iteration of the Simplex Method, the basis is represented as a rooted spanning tree in which an arc (i, j) and its flow x_{ij} represent the basic variable x_{ij} , and the simplex multipliers (dual variables) are represented by *node potentials*. At each iteration, an entering variable (non-tree arc) is selected by some *pricing strategy*. This arc is added to the tree and forms a cycle. The leaving variable is the arc of this cycle with the least augmenting flow. The substitution of entering for leaving arc, and the reconstruction of the tree is called a *pivot*. When no non-basic variable arc remains eligible to enter, the optimal solution has been reached.

The first tree manipulation structure was suggested by Johnson in 1966. The first implementations are those of Srinivasan and Thompson (1973) and Glover, Karney, Klingman and Napier (1974). These papers develop the NSM for the transportation problem, and both result in running times which offer significant reductions over previous solution times for these problems. Since then there has been a great amount of research on the network simplex method for all types of network flow problems.

4.13 THE NETWORK SIMPLEX AND THE TRANSPORTATION PROBLEM

We now describe a simplified version of the NSM as it is applied to the Transportation Problem. We call this method the *Transportation Simplex Method* or TSM. The Transportation problem is simpler than the MCF problem because (i) the network is a bipartite graph which has no transshipment nodes and (ii) has no capacity bounds on the arcs. Although the TSM is simpler than the NSM it contains all of the important aspects of the NSM.

$$\begin{aligned} \text{Minimise} \quad & \sum_{(i,j) \in \mathcal{A}} c_{ij}x_{ij} \\ \text{Subject to} \quad & \sum_{j:(i,j) \in \mathcal{A}} x_{ij} = b_i, \quad \text{for all } i \in \mathcal{N}_1 \\ & \sum_{i:(i,j) \in \mathcal{A}} x_{ij} = -b_j, \quad \text{for all } j \in \mathcal{N}_2 \\ & x_{ij} \geq 0, \quad \text{for all } (i, j) \in \mathcal{A}. \end{aligned}$$

The more standard algebraic formulation is

$$\begin{aligned} \text{Minimize} \quad & \sum_{i=1}^m \sum_{j=1}^n c_{ij}x_{ij} \\ \text{Subject to} \quad & \sum_{j=1}^n x_{ij} = S_i \quad i = 1, 2, \dots, m \quad (\text{Supplies}), \\ & \sum_{i=1}^m x_{ij} = D_j \quad j = 1, 2, \dots, n \quad (\text{Demands}), \\ & x_{ij} \geq 0 \quad \text{for all } i, j, \end{aligned}$$

where $S_i > 0, D_j > 0$ for all i, j and $\sum_{i=1}^m S_i = \sum_{j=1}^n D_j$, i.e., total supply = total demand. This last condition is necessary and sufficient for the problem to be feasible.

A transportation problem is shown graphically in Figure 1.

The interpretation of the network in Figure 1 is this : There are m factories each producing the same product. Factory i produces an amount S_i . There are n warehouses each demanding the same product. Warehouse j demands an amount D_j . x_{ij} is the amount shipped from factory i to warehouse j at a cost of c_{ij} for each unit

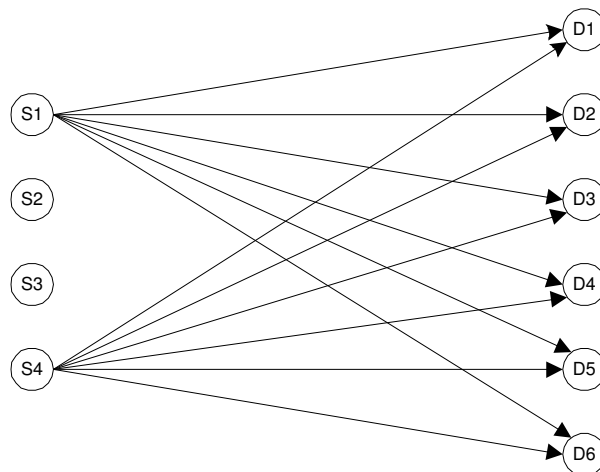


Figure 4.19. Bipartite Graph for Transportation Problem.

shipped. A factory cannot ship out more than it produces. This gives us the supply constraints. For example, the factory 2 constraint is $\sum_{j=1}^n x_{2j} = S_2$. Likewise, a warehouse cannot receive more than it demands. This gives us the demand constraints. For example the warehouse 6 constraint is $\sum_{i=1}^m x_{i6} = D_6$. If supply exceeds demand we 'ship' all excess to a 'dummy' warehouse $n + 1$, at zero cost. If demand exceeds supply the problem cannot be solved as a transportation problem and it needs to be reformulated.

4.13.1 Properties of the Transportation Problem.

The standard form of the general linear program is

$$z = \min_{x \in R^n} cx, \text{ subject to: } Ax = b, x \geq 0,$$

where $b, c,$ and x are $m \times 1, n \times 1,$ and $m \times 1,$ vectors respectively, and A is an $m \times n$ matrix.

For the transportation problem with m supplies and n demands m becomes $m + n$ and n becomes $mn,$ and we have

$$\begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} & c_{21} & c_{22} & \dots & c_{2n} & \dots & c_{m1} & c_{m2} & \dots & c_{mn} \\ 1 & 1 & \dots & 1 & & & & & & & & & \\ & & & & 1 & 1 & \dots & 1 & & & & & \\ & & & & & & & & \dots & & & & \\ & & & & & & & & & 1 & 1 & \dots & 1 \\ 1 & & & & 1 & & & & & 1 & & & \\ & 1 & & & & 1 & & & & & 1 & & \\ & & \ddots & & & & \ddots & & \dots & & & \ddots & \\ & & & 1 & & & & 1 & & & & & 1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ \vdots \\ x_{1n} \\ x_{21} \\ x_{22} \\ \vdots \\ x_{2n} \\ \vdots \\ x_{m1} \\ x_{m2} \\ \vdots \\ x_{mn} \end{bmatrix} = \begin{bmatrix} z \\ S_1 \\ S_2 \\ \vdots \\ S_m \\ D_1 \\ D_2 \\ \vdots \\ D_n \end{bmatrix}$$

The A matrix above is the node-arc incidence matrix for the bipartite graph $G = (\mathcal{N}_1, \mathcal{N}_2, \mathcal{A}, c),$ where $|\mathcal{N}_1| = m, |\mathcal{N}_2| = m,$ and $|\mathcal{A}| = mn$ if G is a complete bipartite graph.

Any linear program with such an A matrix has the following properties :

1. It is a linear programming problem with $m \times n$ variables x_{ij} and $m + n$ constraints.
2. There is exactly one redundant constraint. Any one of the constraints may be treated as redundant and dropped.
3. A basic vector consists of $(m + n - 1)$ basic variables.
4. The $(m + n - 1) \times (m + n - 1)$ basis matrix B is triangular.
5. If all the S_i 's and D_j 's are positive integers, then all basic solutions are integer and hence the optimum solution is integer.
6. The A matrix and possibly the c vector are very sparse. Indeed, the A matrix has $(m + n)mn$ elements and only $2mn$ of these are non-zeros. In addition these non-zeros are all 1's and have a very special structure.

All of these properties contribute to make the transportation problem a very special linear programming problem. For this reason special forms of the simplex algorithm are used to solve it in a more efficient manner than the general simplex algorithm.

4.13.2 The Dual Transportation Problem.

The dual of the transportation problem is important because we use dual variables in the calculation of the reduced costs in the primal problem.

$$\begin{aligned} &\text{Maximize } \sum_{i=1}^m S_i u_i + \sum_{j=1}^m D_j v_j \\ &\text{Subject to: } u_i + v_j \leq c_{ij} \text{ for all } i, j, \\ &\quad u_i, v_j \text{ unrestricted.} \end{aligned}$$

We now show how the dual constraints are used to calculate the reduced costs $\bar{c}_{ij} = c_{ij} - u_i + v_j$ for the non-basic variables. Notice that there is one dual constraint for each arc in the network.

To calculate the reduced costs we need to calculate the simplex multipliers or node potentials u_i and v_j for $i = 1, \dots, m$ and $j = 1, \dots, n$. If x_{ij} is a basic variable then $u_i + v_j = c_{ij}$. This gives us $m + n - 1$ equations in $m + n$ unknowns. Now, because one of the constraints is redundant, we can choose an arbitrary value for any one of the u 's or v 's. Choosing $u_1 = 0$ we solve for the remaining $m + n - 1$ variables by forward or back substitution. This is always possible because the equations can be triangularized (remember that the basis matrix B is triangular). For each non-basic variable we have $u_i - v_j < c_{ij}$ or $u_i - v_j + \bar{c}_{ij} = c_{ij}$. Hence $\bar{c}_{ij} = c_{ij} - (u_i + v_j)$ can be calculated for all non-basic variables, given u_i and v_j .

4.13.3 The Simplex Method.

The main steps of the simplex method for the transportation problem are as follows :

1. Select $(m + n - 1)$ variables x_{ij} to form the initial basic feasible solution.
2. Check if the solution is improved by introducing a non-basic variable. If so, GO THE STEP 3. Otherwise STOP.
3. Determine which basic variable leaves the basis when the variable selected in Step 1 enters the basis.
4. Adjust the values of the other basic variables. GO TO STEP 2.

Initial Basic Feasible Solution. There are many ways of finding an initial basic feasible solution. We will use the “North-West Corner” (NWC) rule for step 1. There are also many ways of carrying out steps 2,3, and 4. Two well-known methods are (i) the “Stepping-Stone” Method, and (ii) the MODI (Modified for Distribution) Method. We will use the MODI method for steps 2,3, and 4. These are simple methods and are used here to illustrate the basic steps of the simplex algorithm for the transportation problem.

We start with a simple example to illustrate the steps. Because of the special structure of the problem we do not use a simplex tableau. Instead, we use what is called a Transportation Table which greatly simplifies the hand calculations. This table has $m + 2$ rows and $n + 2$ columns as shown in Figure 2. Notice that the table contains all information about the problem and the current basic feasible solution.

Example : . Three factories and four warehouses with the following data : $S_1 = 80$, $S_2 = 50$, $S_3 = 90$, $D_1 = 60$, $D_2 = 35$, $D_3 = 85$, $D_4 = 40$. The cost data is shown in Figure 4.2.

		D1	D2	D3	D4	
S1		6	8	4	5	80
S2		2	9	3	8	50
S3		5	4	9	7	90
		60	35	85	40	220

Figure 4.20. A 3×4 Transportation Table.

We use the North-West Corner Rule as follows : Starting at the N.W. cell in the transportation table, allocate as much as possible to this all without violating any constraints. This will result in either the column or row corresponding to this all being fully allocated (column 1 in the example).

Move East or South (right or down) one cell and again allocate as much as possible to this cell without violating any constraints.

Repeat this process until $m + n - 1$ cells have been allocated. Some of these cells may have an allocation of 0. These $m + n - 1$ cells correspond to a basic feasible solution.

The initial basic feasible solution obtained by the N.W. Corner Rule is shown in Figure 4.3.

The basic vector is $(x_{11}, x_{12}, x_{22}, x_{23}, x_{33}, x_{34}) = (60, 20, 15, 35, 50, 40)$

$$\text{Total Cost} = 60 \times 6 + 20 \times 8 + 15 \times 9 + 35 \times 3 + 50 \times 9 + 40 \times 7 = 1490$$

		D1		D2		D3		D4		u
	S1	60	6	20	8	4	5			0
	S2		2	-1	9	1	3	8		1
	S3		5	1	4	-1	9	7		7
	v	6		8		2		0		1490

Figure 4.21. The Initial Basic Feasible Solution by NW Corner Rule.

Improving a Basic Feasible Solution.

Once we have a basic feasible solution we must check to see if the solution can be improved by introducing a non-basic variable, i.e., an unallocated cell. We do this using the ‘MODI’ method which is as follows:

1. Calculate the dual variables u and v corresponding to each constraint (row and column). This is done by setting $u_1 = 0$ and solving $u_i + v_j = c_{ij}$ for each allocated cell.
2. Calculate the reduced costs $\bar{c}_{ij} = c_{ij} - (u_i + v_j)$ for each un-allocated cell.
3. Choose an un-allocated cell (i^*, j^*) with a *negative* reduced cost and find this cell’s *unique θ -loop* as follows : Place a $(+\theta)$ in (i^*, j^*) ; place a $(-\theta)$ or a $(+\theta)$ or nothing in each basic (allocated) cell in such a way that each row and column has a $(+\theta, -\theta)$ pair or nothing. The cells with θ s define the unique θ -loop for (i^*, j^*) .
4. Find the $(-\theta)$ cell with the smallest flow. Let this be cell (i, j) and let $\Delta f = x_{ij}$. Add Δf to all $(+\theta)$ cells and subtract Δf from all $(-\theta)$ cells. This decreases x_{ij} to 0 and increases $x_{i^*j^*}$ to Δf . Thus we have a new basic feasible solution with x_{ij} non-basic and $x_{i^*j^*}$ basic and the other θ -loop basic variables adjusted by $\pm\Delta f$ to maintain balance and hence feasibility.
5. Repeat the steps above until no negative reduced cost is found. The solution is then optimal.

Summary of Transportation Simplex Steps. Combining the two procedures above we get the complete Simplex Algorithm for the Transportation Problem.

1. Construct an initial basic feasible solution (NWC).
2. Calculate the dual variables u_i and v_j , and the reduced costs $\bar{c}_{ij} = c_{ij} - (u_i + v_j)$.
3. Choose a non-basic variable (unallocated cell) $x_{i^*j^*}$ with a negative reduced cost, find its unique θ -loop. If no such variable is found, stop.

- Using the unique θ -loop find the basic variable x_{ij} that leaves the basis. This is the $(-\theta)$ cell with the smallest allocation, Δf . Reallocate Δf to the cell (i^*, j^*) and adjust the allocations around the unique θ -loop. This drives x_{ij} to 0 and it becomes non-basic.

Applying these steps to the problem above we get the sequence of tables shown in Figure 4. The final table is the optimum solution with Total Cost = 920. Alternative optima exist because $\bar{c}_{34} = 0$, i.e. we can change the allocations without changing the total cost.

4.14 THE NETWORK SIMPLEX METHOD

The MODI Method and other methods that use a transportation table are adequate as long as the problem is relatively small. Once the problem becomes large ($m, n > 100$) then finding the unique θ -loop becomes extremely difficult and performing the updating is tedious.

For these and other reasons the standard method of solving transportation and network flow problems has been the *Out-of-Kilter* algorithm for Network Flows, the transportation problem being a special case of the network flow problem. In the late 1950's and early 60's some computational experiments were carried by Dantzig and others which appeared to demonstrate the superiority of the out-of-kilter algorithm. In fact these experiments were very limited and the results were extrapolated to the erroneous conclusion that the out-of-kilter algorithm was best. This conclusion became part of the OR folklore and as a result little or no research was done on specializing the Simplex Method for network flow problems.

Around 1970 research on new methods of solving the transportation problem began to appear. The key idea in this work was to use a compact network representation of the problem along with a specialization of the primal simplex algorithm. Although most of the key network (or graph-theoretical) facts about the transportation and minimum cost flow problems were known (see Dantzig' book, 1963, Chap. 19) and a paper by Ellis Johnson (1965) had explicitly advocated these ideas, most researchers were tied to the algebraic view of these problems. The development of the compact network representation idea was probably inspired by the newly-developed Computer Science techniques of data structures and their manipulation.

This work has resulted in very compact representations so that very large problems can be stored in memory. This, in turn, has meant that the algorithms have less work to do because the problem is compactly represented.

4.14.1 Basis Trees.

If we consider the 3×4 example again we see that each table has a corresponding network. Each arc between a supply node and a demand node represents a variable in the basis. The missing arcs represent nonbasic variables. Each network has the following properties:

- $m + n$ nodes
- $m + n - 1$ arcs
- Each node is connected to all other nodes by 1 or more arcs.
- There are no loops, i.e., there is only one sequence of arcs between any pair of nodes.

These properties define a *spanning tree* of the transportation network which has $m + n$ nodes and $m \times n$ arcs.

The above facts allow us to represent each basis as a spanning tree. This is called the *Basis Tree*. Adding an arc to a basis tree is equivalent to an incoming non-basic variable. The addition of an arc to a basis tree forms a unique cycle and this corresponds to the unique θ -loop of the table. This loop is broken by deleting some other arc in the loop. This corresponds to a basic variable leaving the basis and becoming non-basic.

The structure of each basis spanning tree becomes more clear if we redraw them as follows: Choose the first node (number 1 or S_1 , for this example) as the *root* of the basis tree. Draw all nodes and arcs below the root, as shown in Figure 6.

It should be obvious that all information about a basic feasible solution is contained in its basis tree. The only other information needed to move from one basis to the next is a list of the non-basic arcs.

	D1	D2	D3	D4	u
S1	60	20		5	0
S2	2	9	50	8	-10
S3	5	15	35	40	-4
v	6	8	13	11	1325

	D1	D2	D3	D4	u
S1	60	0	20	5	0
S2	2	9	3	8	-1
S3	1	5	4	7	5
v	6	-1	4	2	1145

	D1	D2	D3	D4	u
S1	45	3	35	5	0
S2	2	9	3	8	-1
S3	15	5	4	7	-1
v	6	5	4	8	1055

	D1	D2	D3	D4	u
S1	5	3	35	40	0
S2	1	2	50	3	8
S3	55	5	4	9	7
v	6	5	4	5	935

	D1	D2	D3	D4	u
S1	3	6	40	40	0
S2	5	2	45	3	8
S3	55	5	9	4	7
v	3	2	3	0	920

Figure 4.20. Solution of a 3×4 Transportation Problem.

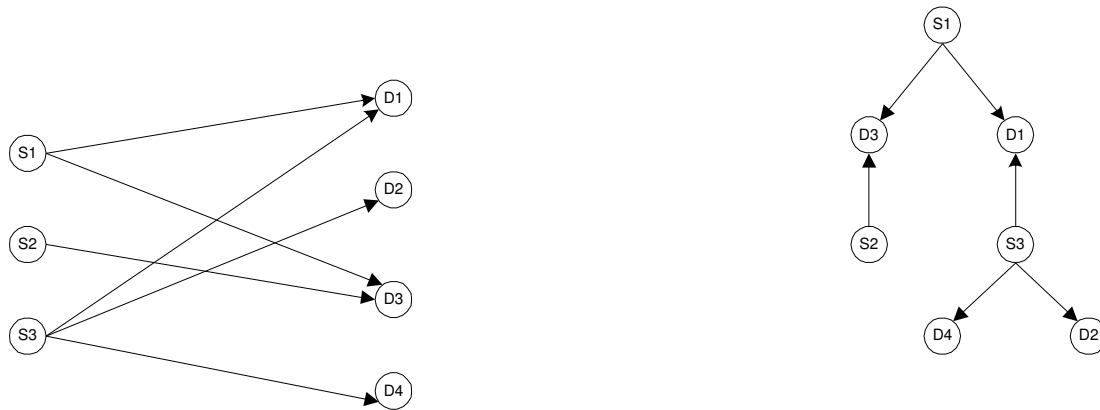


Figure 4.21. A Transportation Network and its Basis Tree.

4.14.2 Pivoting using Basis Trees.

Once a non-basic variable is selected to enter the basis the process of transforming the current basis to the new basis is called *pivoting*. We now show how to pivot using basis trees. In this process all algebraic operations are replaced by graph operations. This not only clarifies the operations but greatly simplifies the computational steps.

Summary of Basis Tree Pivoting. Given that a non-basic arc (i^*, j^*) with a negative reduced cost has been selected from the arc list, the following steps are performed :

1. Add this new arc between node i^* and node j^* . This will form a unique θ -loop which is found as follows:
 - (a) Starting at node i^* climb up the tree to the *root*. This gives a unique path $i^* \rightarrow \dots \rightarrow \text{root}$.
 - (b) Starting at node j^* climb up the tree to the *root*. This gives a unique path $j^* \rightarrow \dots \rightarrow \text{root}$.
 - (c) These two paths will meet at some node *NCA*, the *nearest common ancestor* of i^* and j^* . The unique θ -loop is $i^* \rightarrow \dots \rightarrow \text{NCA} \rightarrow \dots \rightarrow j^* \rightarrow i^*$
2. Using the unique θ -loop find the arc that is to be removed from the loop. This is the arc marked $(-)$ with the smallest flow (allocation)
3. Cut the arc found in 2 and reconstruct the tree.
4. Adjust the flow values of the arcs in the unique θ -loop. Calculate the dual variables u_i, v_j for each node in the tree, with $u_1 = 0$.

We can see that all operations involved in pivoting are done on the basis tree. No other part of the network is affected. This greatly reduces the computation and storage required : instead of algebraically operating on a matrix of size $(m + n) \times mn$ we graphically operate on a basis tree of size $m + n$. This is the key idea in the Network Simplex Algorithm and similar algorithms.

Example using Basis Trees. We now solve the previous example using basis trees.

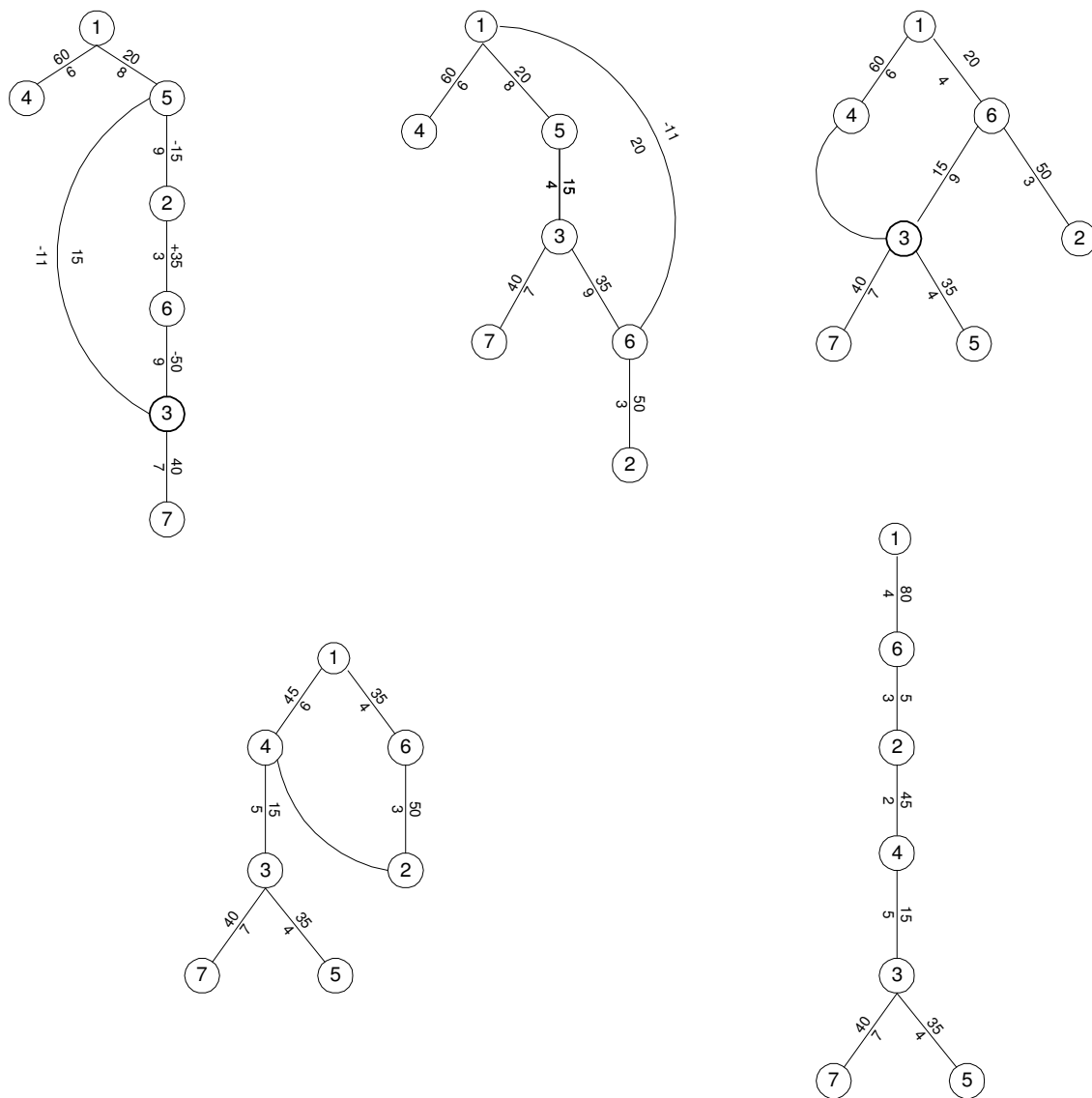


Figure 4.22. Pivoting Using Basis Trees.

4.14.3 The Network Simplex Algorithm.

We now summarize all the steps in the Network Simplex Algorithm. The algorithm has three main procedures : (1) Finding an initial basic feasible solution, (2) Selecting a non-basic arc, and (3) Pivoting. These are combined to form the following outline algorithm.

```

algorithm NetworkSimplex ( $G$ )


---


 $T := \text{InitialBFS}(G)$ 
 $(i^*, j^*) := \text{SelectArc}(G)$ 
while  $(i^*, j^*) \neq \text{NULL}$  do
     $\text{Pivot}(i^*, j^*, T)$ 
     $(i^*, j^*) := \text{SelectArc}(G)$ 
endwhile
return  $(T)$ 
endalg NetworkSimplex

```

Note that the *optimality test* is contained in the *SelectArc* procedure : it returns NULL when it can find no non-basic (non-tree) arc to improve the solution.

The three procedures used in this algorithm are fundamentally different and many different techniques have been proposed for each. There is general agreement on how the *Pivot* procedure should be performed and the various proposals differ only in minor technical details. More important, the *Pivot* procedure does not affect the number of iterations performed. The *SelectArc* procedure however, profoundly affects the the number of iterations and there is much discussion and, indeed, controversy about the best selection strategy to use. In general, a strategy that gives a low iteration count requires a lot of time. Hence the most successful selection strategies try to balance the cost of selection against the cost of iterations. There is very little known about good techniques for *InitialBFS*. In general, the simpler ones seem to be best.

The SelectArc Procedure. This procedure scans the list of arcs in the network, calculating the reduced cost of each arc, and selecting a negative reduced cost arc. If no such arc can be found then the current basic feasible solution is optimal. It ignores arcs that are in the basis.

The main question here is : how should the non-basic arcs be scanned and selected? One strategy is to start scanning where the previous scan stopped and select the first non-basic arc with a negative reduced cost. This is simple and fast but gives many iterations of the algorithm. Another strategy is to select the non-basic arc with the most negative reduced cost. This is simple and slow but tends to minimize the number of iterations of the main algorithm. These strategies are at opposite ends of a range of strategies.

A selection strategy due to Grigoriadis (1988?) scans a block of contiguous arcs and selects the most negative arc. The scanning starts at the end of the previous scanned block. If no negative arc is found in the current block the next block is scanned, and so on. The block size b is usually a small fraction of the arc list size and so we get a combination of fast scanning and low iteration count. The block size b can be tuned for different classes of problems. This selection strategy is elegant, easy to implement, and seems to work well in practice. (See Kelly & O'Neill, 1991, for other selection strategies.)

The Pivot Procedure.. The pivot procedure is the most complicated of the three simplex procedures. In section 5.2 we gave a summary of the steps required. We now formally state these and leave the details of their implementation until section 6. Remember that the pivoting is performed on the basis tree only.

```

procedure Pivot ( $i^*, j^*, T$ )
     $k := \text{NCA}(i^*, j^*, T)$  {nearest common ancestor of  $i^*$  and  $j^*$ }
     $(i_o, j_o) := \text{ArcOut}(i^*, k, j^*)$ 
     $\Delta f := \text{Flow}(i_o, j_o)$ 
    UpdateTree ( $i^*, k, j^*, i_o, j_o, T$ )
    UpdateFlow ( $\Delta f, i_o, k, j_o, T$ )
    UpdatePots ( $i^*, j^*, T$ )
end Pivot
    
```

It should be noted that each step in the pivot procedure works on only parts of the tree. Procedure *NCA* works on the two paths $i^* \rightarrow \text{root}$ and $j^* \rightarrow \text{root}$. Procedure *ArcOut* works on the loop $i^* \rightarrow k \rightarrow j^*$. Procedure *UpdateTree* works on one of the paths $i^* \rightarrow k$ or $j^* \rightarrow k$. Procedure *UpdateFlow* works on the loop $i^* \rightarrow k \rightarrow j^*$. Procedure *UpdatePots* works on the subtree whose root is either i^* or j^* . In other words, none of these procedures do any unnecessary work.

4.15 IMPLEMENTING THE NETWORK SIMPLEX ALGORITHM

The transportation problem is represented as a *weighted, bipartite graph* $G(\mathcal{N}_1, \mathcal{N}_2, \mathcal{A}, c)$ where \mathcal{N}_1 is a set of supply nodes, \mathcal{N}_2 is a set of demand nodes, \mathcal{A} is a set of directed arcs (i, j) , $i \in \mathcal{N}_1, j \in \mathcal{N}_2$, and c is a set of costs (weights) which are assigned to each arc $(i, j) \in \mathcal{A}$.

A basic feasible solution is represented as a *spanning tree* T of G , which has $m + n$ nodes and $m + n - 1$ arcs. Each arc (i, j) of T represents a basic variable x_{ij} , and each arc of G that is not in T represents a non-basic variable.

4.15.1 Abstract Data Types for the NSM.

In keeping with current good programming practice we base the implementation of the Network Simplex Algorithm on the careful choice of appropriate abstract data types. We need just two abstract data types :

1. *ArcList*
2. *BasisTree*

The *ArcList* ADT is a list of arcs in the network \mathcal{G} , where each arc (i, j) in the list contains the following data fields :

$$\text{arc } (i, j) : [i, j, c_{ij}, \text{basic}].$$

The field *basic* is either *true* or *false* to indicate if the arc (i, j) is in the basis.

The operations are (1) *Input* (G) which reads all the arcs of \mathcal{G} into the arclist G , and (2) *SelectArc*(G) which selects a non-basic arc with negative reduced cost from G . This is actually a family of operations because there are many ways of selecting an arc from G .

The *BasisTree* ADT is a spanning tree of \mathcal{G} whose arcs are basic. Each node i in the tree contains the following data fields :

$$\text{node } i : [x_{ij}, \pi_i],$$

where $j = p(i)$ is the *parent* of i , (i, j) is a basic arc in the tree, x_{ij} is the flow from i to j and π_i is the potential of i .

The operations are :

1. *AddChild* (i, j) : Makes the sub-tree whose root is i a sub-tree of the node j . Thus $p(i)$ becomes j .

2. *DeleteChild* (i) : Detaches the sub-tree whose root is i from its parent $p(i)$.
3. *Traverse* (i) : Visits all nodes in the sub-tree whose root is i .
4. *NCA* (i, j) : Finds the nearest common ancestor of nodes i and j .
5. *ArcIndex* (i) : Gives the position of arc $(i, p(i))$ in the arclist G .

Data Structures for the Arc List and Basis Tree.

All supplies, demands, costs and flows are assumed to be integer-valued. This assumption rarely causes any difficulty in practice.

The supply nodes are numbered $1, \dots, m$ and demand nodes are numbered $m + 1, \dots, m + n$.

The supplies and demands are stored in a single integer array of integers

$$\text{Supply} : \text{Supply}[1..MaxSup]$$

The arc list is stored as a single array of records $ArcList[1..MaxArc]$. Each record has the following fields :

$$\text{ArcList} : [Node\ i, Node\ j, Cost\ ij, Basic].$$

The basis tree is stored as a single array of records $Tree[1..MaxNodes]$. Each record has the following fields :

$$\text{Tree} : [LMChild, Parent, LSibling, RSibling, Flow, Pot, ArcIndex].$$

A typical node is shown in Figure ? The first 4 field contain structural information and allow access up, down and across the tree. The ArcIndex field contains the position of the arc $(i, Parent(i))$ in the arclist G .

Operations on the Arc List and Basis Tree.

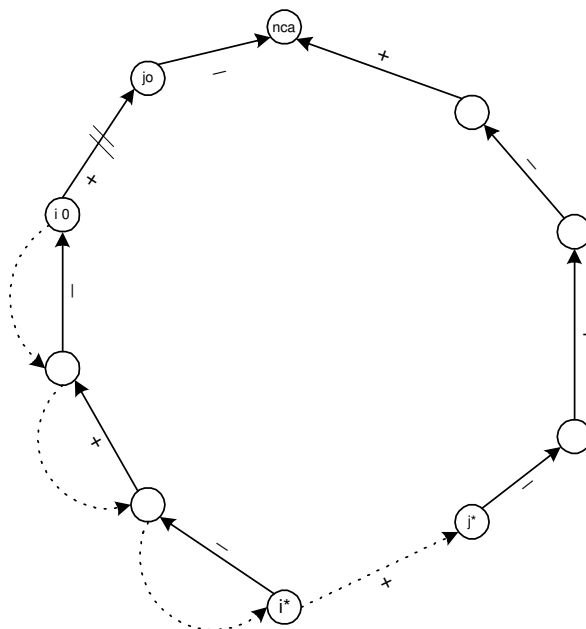


Figure 4.23. Updating Flows and Basis Tree.

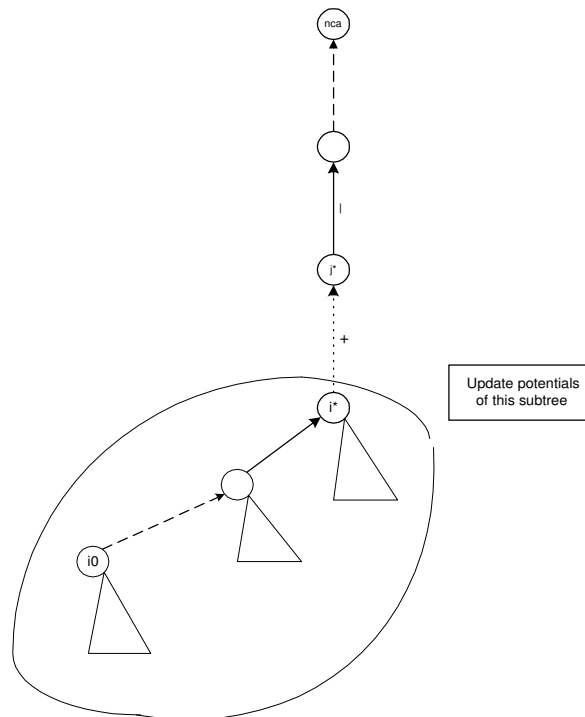


Figure 4.24. Updating Subtree Potentials.

4.16 COMPUTATIONAL EXPERIENCE.

The following results were taken from a M.Mangt.Sc. dissertation by Kelly and O'Neill, 1991. The tests were performed on a 20 MHz 386 Compaq and the Turbo Pascal 5.5 code was compiled with with stack- and range-checking off.

TABLE I — PROBLEM CLASSES

<i>Class</i>	<i>No. of Problems</i>	<i>n</i>	<i>Range of m</i>	<i>Cost Range</i>	<i>Total Supply</i>
Tran	5	200	1398-2900	1-100	1000
Tran	5	300	3196-6370	1-100	1000
Ass	5	400	1619-4786	1-100	200
MCF	4	400	2961-3894	1-100	4000
MCF	4	400	3124-4806	1-100	4000

TABLE II — AVERAGE NO. OF ITERATIONS

<i>Class</i>	<i>Most Negative</i>	<i>First Negative</i>	<i>Arc Block</i>	<i>Arc Sample</i>	<i>Two Phase</i>	<i>Mulvey List</i>	<i>BBG Queue</i>
A	311	1711	481	503	482	565	923
B	505	3464	747	766	754	962	1864
C	572	3181	780	806	815	933	1643
D	99	475	214	214	215	191	313
E	193	885	348	327	341	326	612
F	28	134	53	71	82	52	74

TABLE III — SOLUTION TIMES (SECS)

<i>Class</i>	<i>Most Negative</i>	<i>First Negative</i>	<i>Arc Block</i>	<i>Arc Sample</i>	<i>Two Phase</i>	<i>Mulvey List</i>	<i>BBG Queue</i>
Tran	42.33	23.32	8.17	8.64	8.83	10.76	18.44
Tran	172.14	66.19	21.70	23.24	24.89	28.66	49.28
Ass	121.71	44.85	16.24	17.10	17.28	19.48	33.07
MCF	17.62	5.13	4.17	3.48	3.46	5.90	11.66
MCF	39.82	11.35	7.96	6.62	6.66	9.54	18.04
<i>Total</i>	397.86	152.60	59.37	60.27	62.43	76.68	135.62

The following results are from Barr & Turner [B&T81] who designed and implemented a primal network program for the U.S. Department of the Treasury who use it on a production basis as part of their tax policy model of the U.S. The table below gives that data for a problem with 25,000,000 variables and 110,000 constraints which was solved as 6 subproblems.

Sub-Problem No	1	2	3	4	5	6	Total
No. of Variables	4,703,000	4,832,000	4,723,000	3,680,500	3,743,500	3,317,500	25,000,000
No. of Constraints	15,000	18,000	25,000	14,000	15,500	23,000	110,500
Soln Time (mins)	285	382	780	254	323	598	2,622
(hours)	(4.75)	(6.36)	(13.01)	(4.24)	(5.39)	(9.97)	(43.7)

Calculations were performed on a UNIVAC1108 using FORTRANV level IIA. The program has a design limit of 65,000,000 variables and 50,000 constraints for a single problem.

4.17 CONCLUSION

It is important to realize at this stage that no new algorithm has been proposed. All the algorithms we have discussed or mentioned (Stepping Stone, MODI, Out-of-Kilter, Dual) are variations of the primal or dual Simplex methods : move from extreme point to extreme point on the convex hull until the optimum is reached. The real difference between the latest methods and the older methods is due to the great improvements in problem/data representation and the exploitation of the subtle interplay between the algorithms and the data structures used for their implementation. It is no coincidence that these improvements have taken place along with the development and establishment of computer science as a rigorous discipline.

Current research is focussed on applying the methods above to more general L.P. problems. The standard approach in this research is to try to transform general L.P. problems into network problems. Another approach is to decompose a general L.P. into a general L.P. subproblem and a network subproblem.

The discovery in 1979 of the first polynomial time L.P. algorithm by the Russian mathematician, Khachian is really a Computational Complexity Theory result rather than a break-through in fast L.P. algorithms. Nonetheless, it is a very interesting result, and it has led to many theoretical developments. It also prompted Karmarkar (1984) to develop his interior point L.P. algorithm which appears to be competitive with the primal Simplex algorithm. This has led to a flurry of research activity in non-simplex interior methods. There are many claims and counter-claims for various L.P. algorithms and it is difficult to say which one is best. Probably none is best for all problem classes.

4.18 A SHORT ANNOTATED BIBLIOGRAPHY

Books.

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B : *Network Flows*, Prentice-Hall, 1993. The best book in the field by three of its leading researchers. Contains the latest research and has numerous applications.
2. Bertsekas, D. : *Linear Network Optimization*, MIT Press, 1991. This contains the author's latest work on his *relaxation* and *auction* algorithms for network flows.
3. Chvatal, V : *Linear Programming*, Freeman, 1983. One of the best books on Linear Programming with excellent chapters on Network Flows.
4. Dantzig, G., *Linear Programming and Extensions*, Princeton University Press, 1963. This book is still an excellent source of sophisticated L. P. techniques and problems.
5. Murty, K. G., *Linear and Combinatorial Programming 2nd Ed*, Wiley, 1986. A very good Linear Programming book. It is rigorous, up to date and contains a chapter on the latest transportation methods. Undergrad-graduate level.
6. Lawler, R. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, Winston, 1976. This is a graduate-level text by one of the leaders in the field. Contains most of the modern research and views on simplex-based network algorithms.
7. Papadimitiou, C., and Steiglitz, K. : *Combinatorial Optimization : Algorithms and Complexity*, Prentice-Hall, 1982.
8. Aho, A.V. and Ullman, J.D. : *Foundations of Computer Science*, Computer Science Press (Freeman), 1992.
9. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. : *Data Structures and Algorithms*, Addison-Wesley, 1984.
10. Sedgewick, R. : *Algorithms*, 2nd Ed., Addison-Wesley, 1988.
11. Standish, T.A. : *Data Structures, Algorithms, and Software Principles*, Addison-Wesley, 1994.
12. Tarjan, R.E. : *Data Structures and Network Algorithms*, SIAM, 1983.

Papers.

1. Barr, R. S. and Turner, J. S., "Microdata File Merging Through Large-Scale Network Technology", *Mathematical Programming Study 15*, 1–22, North-Holland, 1981.
2. Bradley, G. H., "Survey of Deterministic Networks," *AIIE Transactions*, Vol. 7, No. 3, September 1975.
3. Bradley, G. H., Brown, G. G., Graves, G. W., "Design and Implementation of Large-Scale Primal Transshipment Algorithms", *Management Science*, Vol. 24, No. 1, September 1977.
This is an excellent and detailed paper with an extensive list of references. [BRA75] also has an extensive list.
4. Glover, F., Karway, D., Klingman, D., & Napier, A., "A Computational study on Start Procedures, Basic Change Criteria and Solution Algorithms for Transportation Problems", *Management Science*, Vol. 20, No. 5, Jan 1974.
The most complete and revealing computational study of its kind. Undermines a lot of folklore.
5. O'Neill, G., and Kelly, D : "The Minimum Cost Flow Problem and the Network Simplex Solution Method", M.Mangt.Sc. Dissertation, MIS Dept, University College, Dublin, Ireland, 1991.