**Assignment 1**

**AIM :** Write a program to find Maximum and Minimum element in an array using Divide and Conquer strategy and verify the time complexity.

**OBJECTIVE:**

-To understand the concept of the Divide and Conquer

**THEORY:**

**FINDING THE MAXIMUM AND MINIMUM**

The problem is to find the maximum and minimum items in a set of n elements.

*1.*      ***Algorithm for straight forward maximum and minimum***

StraightMaxMin(a,n,max,min)

// set max to the maximum and min to the minimum of a[1:n].

{

     max := min := a[1];

     for i := 2 to n do

     {

           if(a[i] > max) then max := a[i];

           if(a[i] > min) then min := a[i];

     }

}

***Analyzing the Straight Forward Method***

                In analyzing the time complexity of this algorithm, we have to concentrate on the number of element comparisons. This algorithm requires 2(n-1) element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing that the comparison a[i] < min is necessary only when a[i]>max is false.

                Now the Best case occurs when the elements are in increasing order. The number of element comparisons is n-1. The worst case occurs when the element are in decreasing order. In this case number of comparisons is 2(n-1).

FINDING THE MAXIMUM AND MINIMUM using DIVIDE AND CONQUER Strategy

         What is DIVIDE AND CONQUER Strategy?

Given a function to compute on n inputs the *divide-and-conquer* strategy suggest splitting the inputs into k distinct subsets, 1 < K ≤ n, yielding k sub problems. These Sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the Sub problems are still relatively large, then the *divide-and-conquer* strategy can possibly be reapplied. Often the sub problems resulting from a *divide-and-conquer* design are the same type as the original problem. For those cases the reapplication of the *divide-and-conquer* principle is naturally expressed by a recursive algorithm. Now smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

A *divide-and-conquer*algorithm for this problem would proceed as follows:  Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list. Let small(P) be true when n ≤ 2. In this case, the maximum and minimum are a[i] if n = 1. If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]). After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

Now the question is How can we combine the Solutions for P1 and P2 to obtain the solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

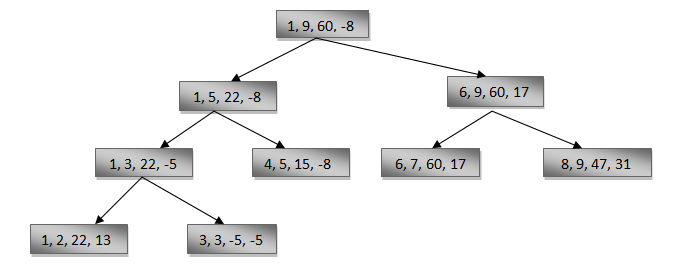
MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of this sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

The procedure is initially invoked by the statement MaxMin(1,n,x,y). for this algorithm each node has four items of information: i, j, max, min. Suppose we simaulate MaxMin on the following nine elements:

a: [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]

    22   13   -5   -8   15  60   17   31  47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.



We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

**2.      Algorithm for maximum and minimum using divide-and-conquer**

**MaxMin(i, j, max, min)**

**// a[1:n] is a global array. Parameters i and j are integers,   // 1≤i≤j≤n. The effect is to set max and min to the largest and  // smallest values in a[i:j].**

**{**

**if (i=j) then max := min := a[i]; //Small(P)**

**else if (i=j-1) then // Another case of Small(P)**

**{**

**if (a[i] < a[j]) then max := a[j]; min := a[i];**

**else max := a[i]; min := a[j];**

**}**

**else**

**{**

**// if P is not small, divide P into sub-problems.**

**// Find where to split the set.**

**mid := ( i + j )/2;**

**// Solve the sub-problems.**

**MaxMin( i, mid, max, min );**

**MaxMin( mid+1, j, max1, min1 );**

**// Combine the solutions.**

**if (max < max1) then max := max1;**

**if (min > min1) then min := min1;**

**}**

**}**

**Time Complexity: O(n)**

**INPUT :** Array of numbers

**OUTPUT:** Maximum and Minimum Number from given array

**FAQS:**

1. What is divide and conquer strategy?
2. Explain the recurrence relation for divide and conquer .
3. Explain the complexity of Algorithm for maximum and minimum using divide-and-conquer

**Assignment 2**

**AIM :** Write a program to solve optimal storage on tapes problem using Greedy approach.

**OBJECTIVE:**

* To understand the concept of Greedy Approach
* To find minimum MRT for tape.

**THEORY:**

*The problem*: Given *n* programs to be stored on tape, the lengths of

these *n* programs are *l1, l2 , . . . , ln* respectively.

Suppose the programs are stored in the order

of *i1, i2 , . . . , in*

Let *tj* be the time to retrieve program *ij*.

Assume that the tape is initially positioned at the beginning.

*tj is proportional to the sum of all lengths of programs stored in front of the program ij.*

Using Greedy approach:

We want to store files of lengths {12,34,56,73,24,11,34,56,78,91,34,91,45} on three tapes. How should we store them on the three tapes so that the mean retrieval time is minimized?

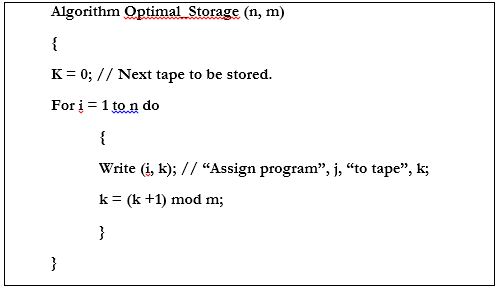
SOLUTION:

1. STORE FILES BY NONDECREASING LENGTH.

First sort the files in increasing order of length. For this we can use meregesort algorithms. 11 12 24 34 34 34 45 56 56 73 78 91 91

2. Now distribute the files:

**Algorithm:**



**Time Complexity: O (n log n)**

**INPUT :** Number of files, number of tape, length of file

**OUTPUT:** MRT of each tape an average MRT

**FAQS:**

1. What is Greedy approach?

2. Explain the complexity of Algorithm.

**Assignment 3**

**AIM :** Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity.

**OBJECTIVE:**

* To understand the concept of the Dynamic programming.
* To understand the concept of Principal of Optimality.

**Theory:**

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.

Dijksra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra

**Algorithm:**

***BELMAN-FORD***( G, s )

***INIT***( G, s )

for i ←1 to |V|-1 do

for each edge (*u*, *v*) ∈ E do

***RELAX***( u, v )

for each edge ( u, v ) ∈ E do

if d[v] > d[u]+w(u,v) then

return *FALSE* **>** neg-weight cycle

return *TRUE*

**1)** This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.  
…..**a)** Do following for each edge u-v  
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]  
………………….dist[v] = dist[u] + weight of edge uv

**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v  
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”  
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**Relax Algorithm**

**relax(u, v):**

**if v.distance > u.distance + weight(u, v):**

**v.distance = u.distance + weight(u, v)**

**v.p = u**

**Time Complexity: O (VE)**

**INPUT :** Graph and a source vertex

**OUTPUT:** Shortest distance to all vertices from source. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**FAQS:**

1. What is Dynamic Programming approach?

2. Explain the complexity of Bellman-Ford Algorithm.

**Assignment 4**

1. Write a program to solve the travelling salesman problem and to print the path and the cost using Dynamic Programming.

**OBJECTIVE:**

* To understand the concept to find the shortest path using travelling sales man problem

**Theory**

Problem Statement:

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

**C(S,j)=minC(S−{j},i)+d(i,j) wherei∈Sandi≠j**

**Algorithm: Traveling-Salesman-Problem**

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

**Time Complexity: O (n22n)**

**INPUT:** Graph and a source vertex

**OUTPUT:** shortest possible route that he visits each city exactly once and returns to the origin city

**FAQS:**

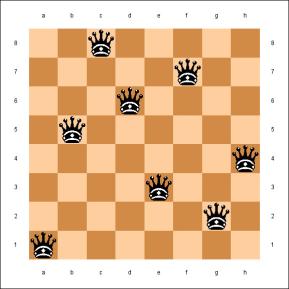
1. Explain the time complexity of TSP.

**Assignment 5**

**AIM :** Write a recursive program to find the solution of placing n queens on achessboard so that no queen takes each other.

**OBJECTIVE:**

* To understand the n queens problem
* To learn backtracking method.
* To find the solution for the n queens problem. ]



**Theory**

The **eight queens puzzle** is the problem of placing eight chess queens on an 8×8 chessboard so that none of them can capture any other using the standard chess queen's moves. The queens must be placed in such a way that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general ***n*-queens problem**of placing*n*queens on an*n*×*n*chessboard.

**Backtracking Method:**

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

Backtracking can be applied only for problems which admit the concept of a "partial candidate

solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates.

It is therefore a met heuristic rather than a specific algorithm --- although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time.

**Description of the method**

The backtracking algorithm enumerates a set of partial candidates that, in principle, could be completed in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of candidate extension steps.

Conceptually, the partial candidates are the nodes of a tree structure, the potential search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further. The backtracking algorithm traverses this search tree recursively, from the root down, in depth- first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped (pruned). Otherwise, the algorithm (1) checks whether c itself is a valid solution, and if so reports it to the user; and (2) recursively enumerates all sub-trees of c. The two tests and the children of each node are defined by user- given procedures.

Therefore, the actual search tree that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is basically the number of nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.

**Solution to 8 queens problem:**

8 queens problem can be solved using backtracking method. The problem can be generalized. Consider an n×n chessboard and try to find all ways to place n non-attacking queens. Let (x1, …., xn) represent solution in which xi is the column of the ith row where the ith queen placed. The xi’s will all be distinct since no two queens can be placed in the same column.

If we imagine the chessboard squares being numbered as the indices of the two-dimensional array a[1:n, 1:n], then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row – column value. Suppose two queens are placed at positions (I,j) and (k,l). Then by the above they are on the same diagonal only if

i – j = k – l or

I + j = k + l

The first equation implies j – l = i – k

The second implies J – l = k – i

Therefore two queens lie on the same diagonal if and only if | j – l | = | i – k |.

Algorithm Place(k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether I is distinct from all previous values x[1], …., x[k-1] and whether there is no other queen on the same diagonal. Its computing time is O( k-1).

**Algorithm Place(k,i)**

// Returns **true** if a queen can be placed in kth row and ith column. Otherwise it returns **false**.

//X[] is a global array whose first (k-1) values have been set. //Abs( ) returns absolute value of r

**{**

**for** j := 1 to k-1 **do**

**if** (( x[j] =i) //Two in the same column

**or** (Abs(x[j]–i) = Abs(j–k))) // or in the same diagonal **then return false**;

**return true;**

**}**

Using Place, we can give a precise solution to n- queens problem. Algorithm NQueens is the solution to the problem.

The array x[ ] is global. The algorithm is invoked by NQueens(1,n);

**Algorithm NQueens(k,n)**

//Using backtracking, this procedure prints all possible placements of n queens on an n×n chessboard so that they are nonattacking.

**{**

**for** i:= 1 **to** n **do**

**{**

i**f** Place(k,i) **then**

**{**

x[k] := I;

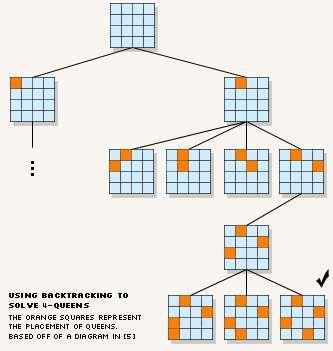
**if** (k==n) **then write** (x[1:n]); **else** NQueens(k+1,n);

**}**

**}**

**}**

For an 8×8 chessboard there are (64 8) possible ways to place 8 pieces, or approximately 4.4 billion 8- tuples to examine. However, by allowing only placements of queens on distinct rows and columns, we require the examination of the most 8!, or only 40,320 8-tuples.



**Figure: 1 – queens solution space**

We can use Estimate to estimate the number of nodes in the solution space that will be generated using NQueens.

**Algorithm Estimate( )**

//This algorithm follows a random path in a state space tree and produces an estimate of the number of nodes in the tree.

**{**

k := 1; m := 1; r :=1;

**repeat**

***{***

Tk := {x[k] | x[k] ε T( x[1], x[2], …, x[k-1]) **and** Bk(x[1],…, x[k]) is true};

**if** (Size(Tk) = 0) **then return** m;r := r \* Size(Tk); m:= m + r; x[k] := Choose(Tk); k:= k + 1;

**} until (false);**

**}**

The assumptions that are needed for Estimate do hold for NQueens. The bounding function is tatic. No change is made to the function as the search proceeds. In addition, all nodes on the same level of the state space tree have the same degree. In following figure, five 8×8 chessboards are shown that are crated using Estimate.

**INPUT:**

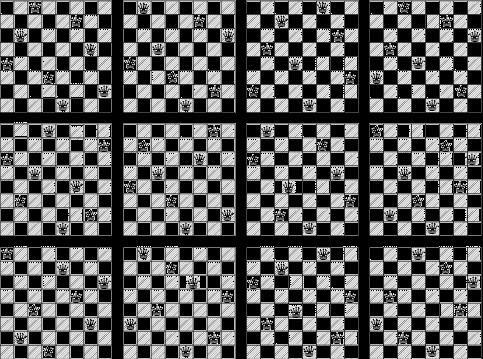
Number of queens on the chessboard (eg n = 8)

**OUTPUT:**

The 8 queen puzzle has 92 solutions.

If the solutions that differ by only

symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 unique solutions.



**FAQS:**

1. What is n queens problem?
2. What is solution space?
3. Explain solution space for 4- queens problem.
4. Explain backtracking method
5. Differentiate between brute force method and backtracking method.

**Assignment 6**

**AIM:** Write a program to solve the travelling salesman problem. Print the pathand the cost using Branch and Bound.

**OBJECTIVE:**

* To learn branch and bound method
* To understand travelling salesman problem
* To implement travelling salesman problem using branch and bound method

**THEORY:**

A branch-and-bound algorithm consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidate s are discarded, by using upper and lower estimated bounds of the quantity being optimized. The Branch and Bound strategy divides a problem to be solved into a number of sub-problems. It is a system for solving a sequence of subproblems each of which may have multiple possible solutions and where the solution chosen for one sub problem may affect the possible solutions of later sub-problems.

Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all.

Let S be some subset of solutions.

L(S) = a lower bound on the cost of any solution belonging to S Let C= cost of the best solution found so far

If C ≤ L(S),there is no need to explore S because it does not contain any better solution.

If C > L(S), then we need to explore S because it may contain a better solution.

**Branch and Bound approach Algorithm:**

**function CHECKBOUNDS(st,des,cost[n][n])** //Cal. the bounds

Global variable: cost[N][N] //the cost assignment. pencost[0] = t

**for** i ← 0, n − 1 do **for** j ← 0, n − 1 do

reduced[i][j] = cost[i][j] **end for**

**end for**

**for j ← 0, n − 1 do**

reduced[st][j] = ∞ end for

**for i ← 0, n − 1 do**

reduced[i][des] = ∞

**end for**

**reduced[des][st] = ∞**

RowReduction(reduced)

ColumnReduction(reduced)

pencost[des] = pencost[st] + row + col + cost[st][des] return pencost[des]

**end function**

**function RowMin(cost[n][n],i)** //Cal. min in the rowmin = cost[i][0]

**for** j ← 0, n − 1 do

**if** cost[i][j] < min thenmin = cost[i][j]

**end if**

**end for return min end function**

**function ColMin(cost[n][n],i) //Cal. min in the col**

min = cost[0][j] **for** i ← 0, n − 1 do

**if** cost[i][j] < min thenmin = cost[i][j]

**end if**

**end for return min end function**

**function ROWREDUCTION(cost[n][n])** //makes row reductionrow = 0

**for** i ← 0, n − 1 dormin = rowmin(cost, i) **if** rmin ≠ ∞ then

row = row + rmin

**end if**

**for j ← 0, n − 1 do**

**if** cost[i][j≠ ∞ thencost[i][j] = cost[i][j] − rmin

**end if end for end for**

**end function**

**function Columnreduction(cost[n][n]) //Makes column reduction** col = 0

**for** j ← 0, n − 1 do

cmin = columnmin(cost, j) **if** cmin ≠ ∞ then

col = col + cmin

**end if**

**for i ← 0, n − 1 do**

**if** cost[i][j] ≠ ∞thencost[i][j] = cost[i][j] − cmin **end if**

**end for end for**

**end function**

**function Main** //main function **for** i **←** 0, n − 1 do

select[i] = 0

**end for rowreduction(cost)**

columnreduction(cost) t = row + col

**while** allvisited(select) ≠ 1 do **for** i ← 1, n − 1 do

**if** select[i] = 0 then

edgecost[i] = checkbounds(k, i, cost)

**end if end for min = ∞**

**for** i ← 1, n − 1 do **if** select[i] = 0 then

**if** edgecost[i] < min thenmin = edgecost[i]

k = i **end if end if**

**INPUT:**

Number of cities and present in the graph and the distance Matrix

**OUTPUT:**

Shortest path for travelling salesman problem which covers all cities present in the graph.

**FAQS:**

1. Explain branch and bound method.
2. Comparison between backtracking and branch and bound method.
3. 3. State space tree for travelling salesman problem using BB
4. Explain First in first out branch and bound and Least cost branch and bound method.

**end for select[k] = 1**

**for** p ← 1, n − 1 docost[j][p] = ∞

**end for**

**for p ← 1, n − 1 do**

cost[p][k] = ∞

**end for cost[k][j] = 1**

rowreduction(cost)

columnreduction(cost) **end while**

**end function**

24

25

26

27

28

**INPUT:**

Number of cities and present in the graph and the distance Matrix

**OUTPUT:**

Shortest path for travelling salesman problem which covers all cities present in the graph.

**FAQS:**

1. Explain branch and bound method.
2. Comparison between backtracking and branch and bound method.
3. 3. State space tree for travelling salesman problem using BB
4. Explain First in first out branch and bound and Least cost branch and bound method.

**Evaluation Of Algorithm:**

The following matrix is the Cost Matrix which shows the distance between the two cities

**Cost Matrix** =

**1**

**2**

**3**

**4**

**5**

**1**

**-**

10

8

9

7

**2**

**10**

-

10

5

6

29

**3**

**8**

10

-

8

9

**4**

**9**

5

8

-

6

**5**

**7**

6

9

6

-

We know that the sum of row minimum gives us the lower bound. Now we have to find the reduced matrix by subtracting the minimum element from every row. So, row minimum will be

31.

**Reduced Matrix** =

**1**

**2**

**3**

**4**

**5**

**1**

**-**

30

3

1

2

0

**2**

**5**

-

5

0

1

**3**

**0**

2

-

0

1

**4**

**4**

0

3

-

1

**5**

**1**

0

3

0

-

In the above reduced matrix there should be a Zero in every row and every column. Now apply the column minimum principle which means for a particular city I will have come into that city from other city. So, now subtracting the column minimum, we get the lower bound 31+1 = 32.

31

**Reduced Matrix** =

**1**

**2**

**3**

**4**

**5**

**1**

**-**

3

0 (2)

2

0 (1)

**2**

**5**

-

4

0 (1)

1

**3**

**0 (1)**

2

-

0 (0)

1

**4**

**4**

0 (1)

2

-

1

**5**

**1**

32

0 (0)

2

0 (0)

-

Now we have to make the assignments Zero’s in the above matrix. If we can not make an assignment at that particular Zero we have to go for the next highest element in that row. If there is same value, we have to calculate the sum of row penalty and column penalty. If the sum of penalties is more then we have to make that assignment because we incur an additional cost if we don’t make that assignment. Now if we don’t have path from 1 to 3 i.e,

X(13) = 0

**(1)**

we have an additional cost of 2 and the lower bound becomes 32+2 = 34. If we have path from 1 to 3 i.e,

X(13) = 1

**(2)**

We eliminate the respective row and column.

**Reduced Matrix** =

**1**

**2**

**4**

**5**

**2**

**5**

-

0

1

**3**

**-**

2

0

1

**4**

**4**

33

0

-

1

**5**

**1**

0

0

-

We have to check whether every row and every column has a zero otherwise we have to subtract

the minimum element from every respective row or column. So, we get

**Reduced Matrix** =

**1**

**2**

**4**

**5**

**2**

**4**

-

0

0

**3**

**-**

2

0

1

**4**

**3**

0

-

0

**5**

34

**0**

0

0

-

Here we made a column reduction for two columns (1 & 5)which gives a lower bound of 32+1+1 = 34.

Again we calculate the penalty for the above matrix.

**Reduced Matrix** =

**1**

**2**

**4**

**5**

**2**

**4**

-

0 (0)

0 (0)

0

**3**

**-**

2

(0)

1 (0)

**4**

**3**

0 (0)

-

0 (0)

**5**

**0 (3)**

35

0 (0)

0 (0)

-

Now if we don’t have path from 5 to 1 i.e,

X(51) = 0

**(3)**

We have an additional cost of 3 and the lower bound becomes 34+3 = 37. If we have path from 5 to 1 i.e,

X(51) = 1

**(4)**

We eliminate the respective row and column. Reduced Matrix =

**2**

**4**

**5**

**2**

**-**

0

0

3

**2**

0

-

**4**

**0**

-

0

Here we a Zero in every row and column. So, the bound remains the same i.e, 34+0 = 34. If we calculate the penalty for the above matrix,

**Reduced Matrix** =

**2**

36

**4**

**5**

**2**

**-**

0 (0)

0 (0)

**3**

**2**

0 (2)

-

**4**

**0 (2)**

-

0 (0)

Now if we don’t have path from 3 to 4 i.e,

X(34) = 0

**(5)**

we have an additional cost of 2 and the lower bound becomes 34+2 = 36. If we have path from 3 to 4 i.e,

X(34) = 1

**(6)**

We eliminate the respective row and column.

**Reduced Matrix** =

**2**

**5**

**2**

**-**

0

**4**

37

**0**

-

Here we a Zero in every row and column. So, the bound remains the same i.e, 34+0 = 34.

Now we have only two possible assignments with Zero penalty cost which would give us a

feasible solution. X(25) = 1

**(7)**

X(42) = 1

**(8)**

On adding all the paths we get,

X(13) + X(34) + X(42) + X(25) + x(51) = 34

**(9)**

38