Tutorial - 9

* multistage graphs:-

i) forward approach:-

```
1.  Algorithm FGraph (G, k, n, p[])
2-  { // The i/p is k-stage graph G=(V,E) with n
3.  //vertices. Indexed in order of stages E is
    // set of edges and c[i,j] is cost of
    // (i,j) p[1:k] is minimum cost path
4.  cost [n] = 0 ;
5.  For j = n-1 to 1 step 1 do
6.  {
7.      // compute cost [j],
8.      // let r be vertex <j,r> is an edge of 'G'
        // c [j,r] + cost [r] is minimum
9.      cost [j] = c [j+r] + cost [r]
10.     d [j] = r ;
11. }
12.     p[1] = 1;
13.     P[k] = n;
14.     for j = 2 to k-1 do.
15.      p[j] = d [p[j-1]];
16. }
```

ii) Backward approach:

```
Algorithm BGraph (G, k, n, p)
// the i/p is a k-stage graph G = (V, E) with
   'n' vertex.
// Indexed in order of staged E is set of edges
```

GoodLuck Page No.
Date

// & c[i,j] is cost of <i,j>, P[1:k] is minimum
// cost path.

```
{   bcost [1] = 0.0;
    for j=2 to n do
    {
     //compute bcost [j];
     // let r be vertex such that <j,r> is
     // an edge of 'G' & bcost [r] + c [r,j] is min.
    bcost [j] = bcost [r] + c [i,j];
    d [j] = r;
    }
    // find min. cost path
    p [1] = 1 ;
    p [k] = n;
    for j = k-1 to 2 do:
     p[j] = d [p[j+1]];
}
```

* All pair shortest path:

```
function All pair( L (1..n, 1..n)) : array [(1..n, 1..n)
array D [1..n, 1..n]

   D = L
   for k = 1 to n do
   for i = 1 to n do
   for j = 1 to n do
   D [i,j] = min (D[i,j], D[i,k] + D[k,j])
   return D;
```

GoodLuck Page No.

Date

* single source shortest path:-

```
1. Algorithm Bellmanford (v, cost, dist, n)
2. // single source / all-destinations shortest paths
3. // with negative edge costs.
4. {
5   for i=1 to n do // initialize dist
6.      dist[i] = cost (v,i);
7.  for i=2 to n-1 do
8.     for each u such that u ≠ v & u has
9.        at least one incoming edge do
10.    for each <i,u> in the graph do
11.       if dist[u] > dist[i] + cost [i,u] then
12.          dist[u] = dist[i] + cost [i,u];
12. }
```

* optimal Binary search tree:-

```
1. Algorithm OBST (p,q,n)
2. // Given n distinct identifiers a1 < a2 < ... < an &
3. // probabilities p[i]; 1 ≤ i ≤ n & q[i], 0 ≤ i ≤ n
4. // this algo computes the cost c[i,j] of
5. // optimal binary search tree tij for
6. // identifiers.
7. {
8.     for i=0 to n-1 do
9.     {
          w[i,j] = q[i];
          r[i,j] = 0;
          c[i,j] = 0.0;
10.    // optimal trees with one node.
```

```
    w[i, i+1] = q[i] + q[i+1] + p[i+1];
    r[i, i+1] = i+1;
    c[i, i+1] = q[i] + q[i+1] + p[i+1];
}
w[n,n] = q[n]; r[n,n] = 0; c[n,n] = 0.0;
for m = 2 to n do // find optimal trees with
                              m nodes
for i = 0 to n-m do
{
    j = i + m;
    w[i,j] = w[i, j-1] + p[j] + q[j];
    // solve 5.12 using knuth's result
    k = find (c, r, i, j);
      // A value of l0 in range r[i, j-1] ≤ l
    c[i,j] = w[i,j] + c[i, k-1] + c[k, j];
    r[i,j] = k;
}
write (c[0,n], w[0,n], r[0,n]);
}
```

```
Algorithm find (c, r, i, j)
{ min = ∞;
    for m = r[i, j-1] to r[i+1, j] do
      if (c[i, m-1] + c[m, j]) < min then
      {
        min = c[i, m-1] + c[m, j]; l = m;
      }
    return l;
}
```

* Backtracking:—

① General method:—

```
// Algorithm Recursive Backtrack (k)
// This scheme describe the backtracking process
// using recursion on entering the first k-1
// values x[1], x[2], ... x[k-1] of the
// solution vector x[1,n] have been assigned.
{
  for each x[k] ∈ T(x[1], ..., x[k-1]) do
  {
     if CB_k (x[1], x[2], ... x[k]) ≠ 0) then
     {
        if (x[1], x[2], ... , x[k] is a path to an
                                     answer node)
          then write (x[1:k]);
        if (k < n) then Backtrack (k+1);
     }
  }
}
```

② Eight queens problem:—

```
// Algorithm 8Queens (k, 8)
// Using backtracking this procedure prints all
// possible placements of 8 queens on an
// 8×8 chess board so that they are
// non attacking.
  {
     for i:=1 to 8 do
     {  if place (k, i) then
          {  x[k] := i;
             if (k=n) then write (x[1:n]);
```

```
        else 8queens(k+1, 8);
      }
   }
}
```

(3) sum of subsets :-

```
// Algorithm sumOfsub (s, k, r)
// find all subsets of w[1:n] that sum to m.
// The values of x[j], 1 ≤ j < k, have
// already been determined, s = Σ_{j=1}^{k-1} w[j].
    x[j] & r = Σ_{j=k}^{n} w[j].
{
    // Generate left child Note. s+w[k] ≤ m since
    // B_{k-1} is true
     x[k] = 1;
    if (s+w[k]=m) then write (x[1:k]);
    // subset found there is no recursive
     call here as w[j] > 0, 1 < j ≤ n.
    else if (s+w[k] + w[k+1] ≤ m)
      then sumofsub (s+w[k], k+1, r-w[k]);
  if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)
    then
   {   x[k] = 0;
        sumofsub (s, k+1, r-w[k]);
   }
}
```

(4) Graph coloring problem :-

```
Algorithm mColoring (k)
// This algo. was formed using the recursive
backtracking scheme. The graph is represented
by its boolean adjacency matrix
G[1:n ; 1:n]
{ repeat
  { // Generate all legal assignments for x[k]
  Next value (k); // Assign to x[k] a legal order
  if (x[k] = 0) then return;
  if (k=n) then //at most m node colors have
          //been used to color n vertices.
    write (x(1:n]);
  else   mColoring (k+1);
  }
  until (false);
}
```

(5) Hamiltanion cycle :-

```
Algorithm Hamiltanion (k)
// This algo uses the recursive algo. of
// backtracking to find all the hamiltanion
// cycles of a graph. The graph is storedasan
// adjacency matrix G[1:n, 1:n] .All
// cycle begin at node 1.
{
repeat
{ // Generate values for x[k].
  Next value (k); //Assign legal next value.
```

```
        if (x[k] = 0) then return;
        if (k = n) then write (x[1:n]);
          else Hamiltonian (k+1);
        }
      until (false);
}
```