III Year II SEMESTER **[2024-25]**

DESIGN AND ANALYSIS OF ALGORITHMS SYLLABUS (R20)

OBJECTIVES:Upon completion of this course, students will be able to do the following:

- **To learn fundamental algorithmic problems.**
- **To understand methods of designing and analyzing algorithms.**
- **To know various designing paradigms of algorithms for solving real world problems.**

List of experiments:

1. Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

2.Write a program to find the optimal profit of a Knapsack using Greedy method.

3 Write a program for Optimal Merge Patterns problem using Greedy Method.

4.Write a program for Single Source Shortest Path for General Weights using Dynamic Programming.

5.Write a program to find all pair shortest path from any node to any other node within a graph.

6.Write a program to find the non-attacking positions of Queens in a given chess board using backtracking.

7.Find a subset of a given set S = {Sl, S2, ...,Sn} of n positive integers, whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

8.Write a program to color the nodes in a given graph such that no two adjacent can have the same color using backtracking.

9.Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using Backtracking principle.

1. Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

To find the maximum and minimum numbers in a given array numbers[] of size n, the following algorithm can be used. Use divide and conquer approach to find the solution.

Divide and Conquer Approach: In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the

maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1, where y is greater than or equal to x.

Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y].

Max−Min(x,y)Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y]numbers[x...y].

Algorithm: Max - Min(x, y)

if $y - x \leq 1$ then

return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

else

(max1, min1):= maxmin(x, $\lfloor((x + y)/2)\rfloor$)

(max2, min2):= maxmin($\lfloor((x + y)/2) + 1)\rfloor$,y)

return (max(max1, max2), min(min1, min2))

Analysis

Let ***T(n)*** be the number of comparisons made by Max−Min(x,y)Max−Min(x,y), where the number of elements n=y−x+1.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

Eperiment 1:

Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

Procedure:

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 $n \leq 2$.

If n = 1, the maximum and minimum are a[i].

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.

We can combine the Solutions for P1 and P2 to obtain the solution for P as follows.

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).

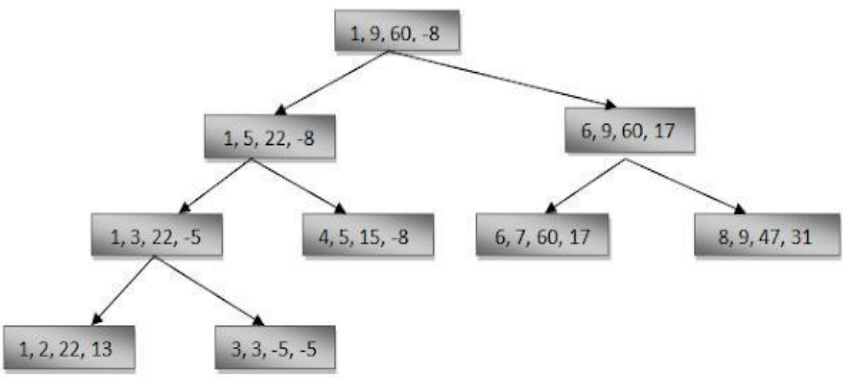
MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When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

Suppose we simulate MaxMin on the following nine elements:

Ex: 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).

Algorithm:

```
Alogrithm 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;
}
}
```

Experiment 2:

Write a program to find the optimal profit of a Knapsack using Greedy method.

Procedure:

Greedy Method:

The greedy method is the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

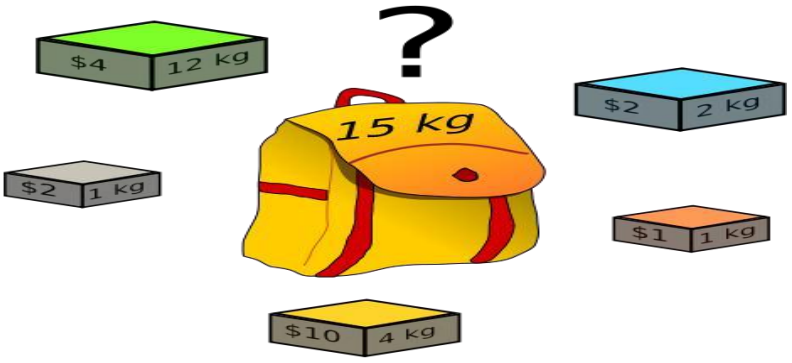
Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Knapsack problem

The knapsack problem or rucksack (bag) problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible



There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to knapsack problems

Brute-force approach:-Solve the problem with a straight forward algorithm

Greedy Algorithm:- Keep taking most valuable items until maximum weight is reached or taking the largest value of each item by calculating $v_i = value_i / Size_i$

Fractional knapsack problem is solved using greedy method as follows

1. For each item, compute its value / weight ratio.
2. Arrange all the items in decreasing order of their value / weight ratio.
3. Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

Example

Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15, 10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x1	x2	x3	wi xi	pi xi
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x2 = 2/15

x1	x2	x3	wi xi	pi xi
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

3. Considering the objects in the order of non-decreasing weights wi.

x1	x2	x3	wi xi	pi xi
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio pi / wi .

p1/w1	**p2/w2**	**p3/w3**
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio pi / xi. Select the object with the maximum pi / xi ratio, so, x2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest pi / xi ratio, so x3 = ½ and the profit earned is 7.5.

x1	x2	x3	wi xi	pi xi
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights
respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
for i := 1 to n do x[i] := 0.0 // initialize x U := m;
for i := 1 to n do
{
if (w(i) > U) then break;
x [i] := 1.0; U := U – w[i];
}
if (i < n) then x[i] := U / w[i];
}
```

Experiment 3:

Write a program for Optimal Merge Patterns problem using Greedy Method

- Given ‘n’ sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge ‘n’ sorted files together.
- This type of merging is called as 2-way merge patterns.
- To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together.
- The two-way merge patterns can be represented by binary merge trees.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, f (i) represents the number of records in each file and d (i) represents the depth.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
treenode *lchild;
treenode *rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{
for i := 1 to n – 1 do
{
pt new treenode
(pt ->lchild)= least (list); // merge two trees with smallest lengths
(pt->rchild) least (list);
(pt-> weight)=((pt-> lchild)->weight)+((pt->rchild)->weight);
insert (list,pt);
}
return least (list);
}
```

- An optimal merge pattern corresponds to a binary merge tree with minimum weighted external path length.
- The function tree algorithm uses the greedy rule to get a two- way merge tree for n files. The algorithm contains an input list of n trees.
- There are three field child, rchild, and weight in each node of the tree. Initially, each tree in a list contains just one node.
- This external node has lchild and rchild field zero whereas weight is the length of one of the n files to be merged.

- For any tree in the list with root node t, t = it represents the weight that gives the length of the merged file.
- There are two functions least (list) and insert (list, t) in a function tree.
- Least (list) obtains a tree in lists whose root has the least weight and return a pointer to this tree.
- This tree is deleted from the list.
- Function insert (list, t) inserts the tree with root t into the list.
- The main for loop in this algorithm is executed in n-1 times.
- If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) time and insert (list, t) can be performed in O(n) time. Hence, the total time taken is O (n^2).
- If the list is represented as a minheap in which the root value is less than or equal to the values of its children, then least (list) and insert (list, t) can be done in O (log n) time. In this condition, the computing time for the tree is O (n log n).

Example:

Suppose we are having three sorted files X_1, X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

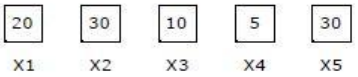
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	X_1 & X_2 = T1	50	T_1 & X_3	60	50 + 60 = 110
2.	X_2 & X_3 = T1	30	T_1 & X_1	60	30 + 60 = 90

The Second case is optimal.

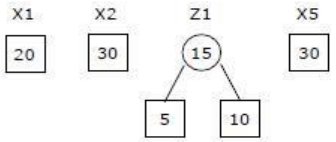
Example 2:

Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

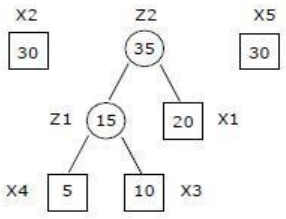
Solution:



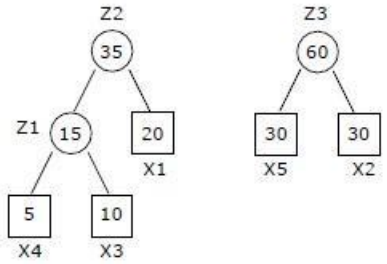
Merge X_4 and X_3 to get 15 record moves. Call this Z_1.



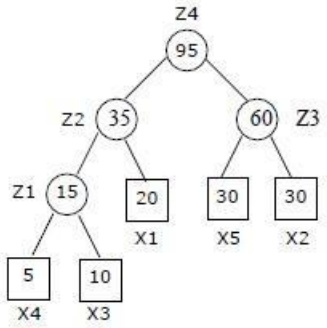
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.



Merge X_2 and X_5 to get 60 record moves. Call this Z_3.



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4.



Therefore the total number of record moves is 15 + 35 + 60 + 95 = 205. This is an optimal merge pattern for the given problem.

Experiment 4:

Write a program for Single Source Shortest Path for General Weights using Dynamic Programming

- Bellman Ford algorithm helps to find the shortest path from a vertex to all other vertices of a weighted graph.
- Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

Procedure:

- Maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.
- We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.
- Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

Algorithm:

```
Algorithm bellmanFord(G,S)
   for each vertex V in G
      distance[V] <- infinite
```

```
        previous[V] <- NULL
    distance[S] <- 0

    for each vertex V in G
        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

Experiment 5:

Write a program to find all pair shortest path from any node to any other node within a graph

- Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the directed and undirected weighted graphs.
- It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).
- This algorithm follows the dynamic programming approach to find the shortest paths.

Algorithm:

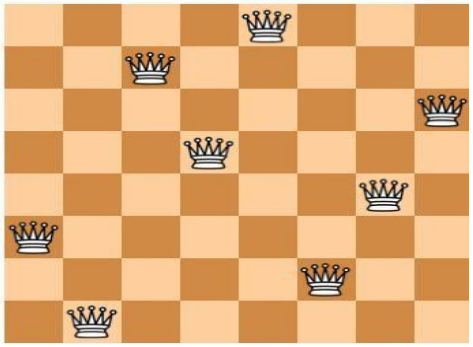
```
Algorithm Floydwarshal()
{
//Initialize the cost matrix
for i := 1 to N
    for j := 1 to N
        if there is an edge from i to j
            dist[0][i][j]:= the length of the edge from i to j
        else
            dist[0][i][j] := INFINITY

for k:= 1 to N
    for I:= 1 to N
        for j:= 1 to N
            dist[k][i][j]:= min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
 }
```

- Initialize the cost matrix same as the input graph matrix.
- update the solution matrix by considering all vertices as an intermediate vertex.
- consider vertex k as an intermediate vertex.
- For every pair (i, j) of the source and destination vertices respectively.

1) k is not an intermediate vertex in shortest path from i to j. no change in the value of dist[i][j].
2) k is an intermediate vertex in shortest path from i to j.
   Update the value of dist[i][j] as
   dist[k][i][j]:= min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])

Week 6:
Write a program to find the non-attacking positions of Queens in a given chess board using backtracking

Problem:
Place N queens on an N x N chessboard so that no two queens attack each other. that is no two queens share the same row, column, or diagonal.
One possible solution when n=8



Algorithm:

- Algorithm place will check whether queen can be placed at k^{th} row and i^{th} column

```
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(r) returns the 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;
}
```

- Algorithm Nqueens place queens in safe positions using backtracking

```
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
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

Week 7:
Find a subset of a given set S = {S_1, S_2, ..., S_n} of n positive integers, whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

Problem:

- Given positive numbers w_i, $1 \le i \le n$, and m, this problem requires finding all subsets of w_i whose sums are „m".
- All solutions are k-tuples, $1 \le k \le n$. Explicit constraints:
  $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$.

Implicit constraints:

No two xi can be the same.

The sum of the corresponding wi" be m.

$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \ge m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \le m$$

$x_i < x_i+1$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

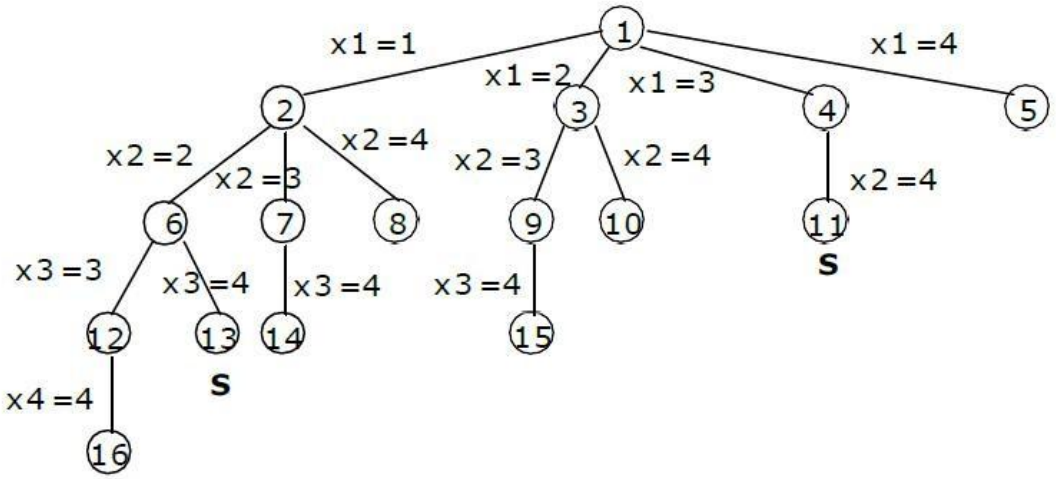
A better formulation of the problem is where the solution subset is represented by an n-tuple (x_1, ... , x_n) such that $x_i \in \{0, 1\}$.

Algorithm:

```
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]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
     // 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]);
     // Generate right child and evaluate B_k.
     if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
     {
          x[k] := 0;
          SumOfSub(s, k + 1, r − w[k]);
     }
}
```

Procedure:
n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11,13, 7) and (24, 7).

- ✓ Solution to the problem constructed by considering the each element in the set.
- ✓ Decision is made either to select or reject the element.
- ✓ If the element is selected it is added to the partial sum.
- ✓ If it is rejected then it is not added to the partial solution.
- ✓ Constraint is verified after selecting an element if constraint violated algorithm backtrack to consider the next alternative.
- ✓ If constraint satisfied algorithm proceeds to make decision on next element.



The tree corresponds to the variable tuple size formulation.
The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i.
At each node, the solution space is partitioned into sub - solution spaces.
All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.
The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2,3), and so on.
Thus, the left mot sub-tree defines all subsets containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, and so on.

Week 8:
Write a program to color the nodes in a given graph such that no two adjacent can have the same color using backtracking.

Graph coloring problem:

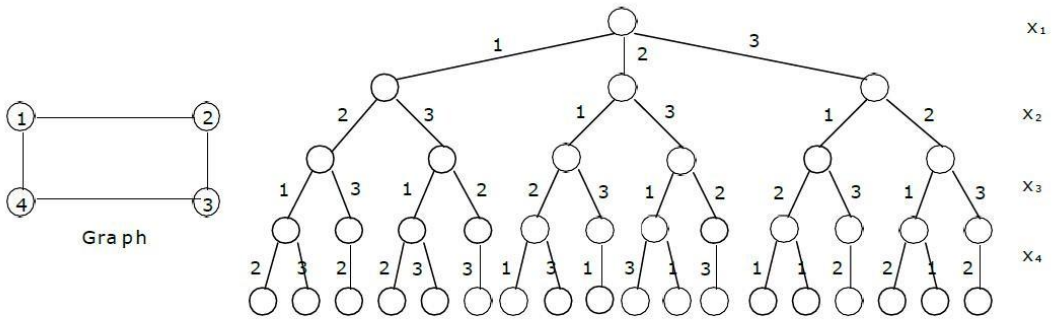
- ✓ Let G be a graph and m be a given positive integer.
- ✓ Find whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used.
- ✓ This is termed the m-colorabiltiy decision problem.
- ✓ The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.
- ✓ The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero.
- ✓ The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.
- ✓ A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

Algorithm:

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, . . . , m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

Example:



A 4-node graph and all possible 3-colorings

Week 9:

Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using Backtracking principle.

Problem:

- ✓ Given graph G with n vertices and E edges

  Find all possible Hamiltonian cycles in a graph G.

- ✓ The Hamiltonian cycle is the cycle that traverses all the vertices of the given graph G exactly once and then ends at the starting vertex.
- ✓ The input can be the directed or undirected graph.
- ✓ Graph G is represented as boolean valued adjacency matrix

  Where G[i,j]= 1 if there exist an edge between i,j

  = 0 otherwise

Backtracking Solution:

- ✓ Problem involves checking if the Hamiltonian cycle is present in a graph **G** or not.
- ✓ Following bounding functions are to be considered while generating possible Hamiltonian cycles
  1. The k^{th} visiting vertex in the path must be adjacent to the $(k-1)^{th}$ visiting vertex in any path (G[x[k-1],x[k]]==1).
  2. The starting vertex and the n^{th} vertex should be adjacent (G[x[n],x[1])==1.
  3. Each vertex should be visited only once.

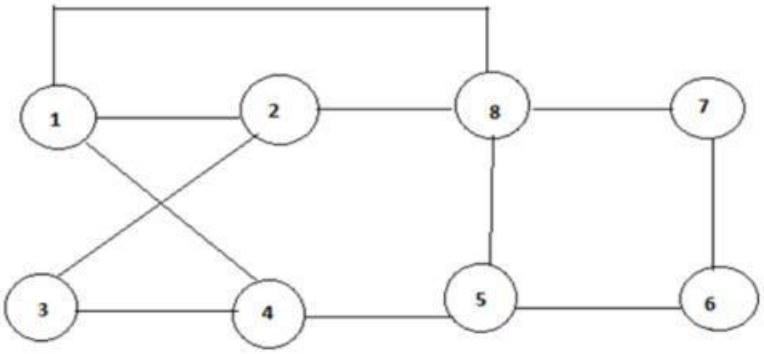
Algorithm:

```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                                  // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

- ✓ This algorithm uses the recursive formulated of backtracking to find all the Hamiltonian cycles of a graph.
- ✓ The graph is stored as an adjacency matrix g[1: n, 1: n].
- ✓ In the next value k, x [1: k-1] is a path with k-1 distinct vertices.
- ✓ if x[k] == 0 then no vertex has to be yet been assign to x[k].
- ✓ After execution x[k] is assigned to the next highest numbered vertex which does not already appear in the path x [1: k-1] and is connected by an edge to x [k – 1] otherwise x[k] == 0.
- ✓ If k == n, (here n is the number of vertices) then in addition x[n] is connected to x [1].

Example:



Hamiltonian cycle 1 8 7 6 5 4 3 2 1