RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **CS-402**

Semester: **4th**

Unit-4

Syllabus: Backtracking concept and its examples like 8 queen's problem, Hamiltonian cycle, Graph coloring problem etc. Introduction to branch & bound method, examples of branch and bound method like traveling salesman problem etc. Meaning of lower bound theory and its use in solving algebraic problem, introduction to parallel algorithms.

BACKTRACKING:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, ... , xn) where each xi Є S, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x1, ... , xn). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Terminology:

- **Problem state** is each node in the depth first search tree.
- **Solution states** are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.
- **Answer states** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.
- **State space** is the set of paths from root node to other nodes. State space tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.
- **Live node** is a node that has been generated but whose children have not yet been generated.
- **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
- **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.
- Depth first node generation with bounding functions is called backtracking. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

N-QUEENS PROBLEM:

The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal.
All solutions to the 8-queens problem can be represented as 8-tuples (x1. . . x8), where xi is the column of the i_{th} row where the i_{th} queen is placed.
The promising function must check whether two queens are in the same column or diagonal:
Suppose two queens are placed at positions (i, j) and (k, l) Then:

- **Column Conflicts:** Two queens conflict if their xi values are identical.
- **Diagonal conflict:** Two queens i and j are on the same diagonal if: i – j = k – l.
  This implies, j – l = i – k
- **Diagonal conflict:** i + j = k + l. This implies, j – l = k – i

Algorithm:

1) Start in the leftmost column
2) If all queens are placed return true
3) Try all rows in the current column. Do following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row, column] as part of the Solution and recursively check if placing queen here leads to a solution.
   b) If placing queen in [row, column] leads to a solution then return true.
   c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
4) If all rows have been tried and nothing worked, return false to trigger backtracking.

8-QUEENS PROBLEM:

The eight queen's problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal).

Algorithm for new queen be placed	All solutions to the n·queens problem

Algorithm Place(k,i)
//Return true if a queen can be placed in kth row & ith column
//Other wise return false
{
for j:=1 to k-1 do
if(x[j]=i or Abs(x[j]-i)=Abs(j-k)))
then return false
return true
}
	Algorithm NQueens(k, n)
	// its prints all possible placements of n-queens on an n×n chessboard.
	{
	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);
	}
	}}

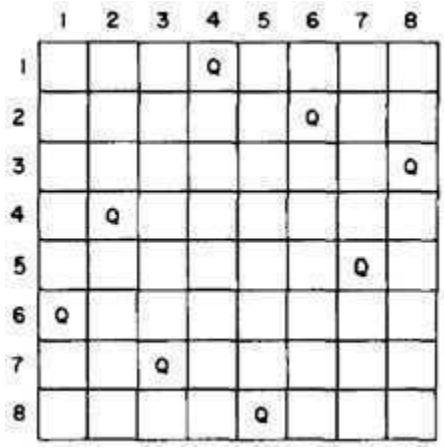

Figure-4.1: 8-Queens Solution

HAMILTONIAN CYCLES:

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position.

In graph G, Hamiltonian cycle begins at some vertiex v1 ∈ G and the vertices of G are visited in the order $v_1, v_2, \text{---} v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$.

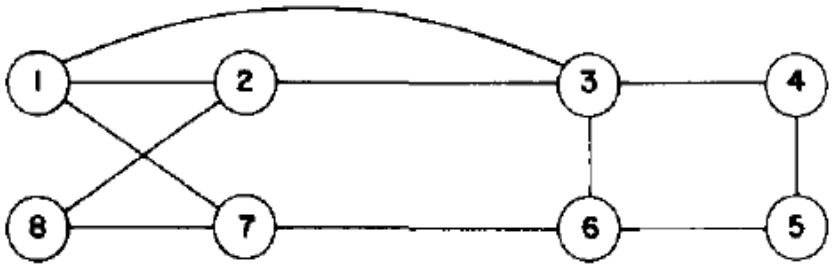

Figure-4.2: Example of Hamiltonian cycle

The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1

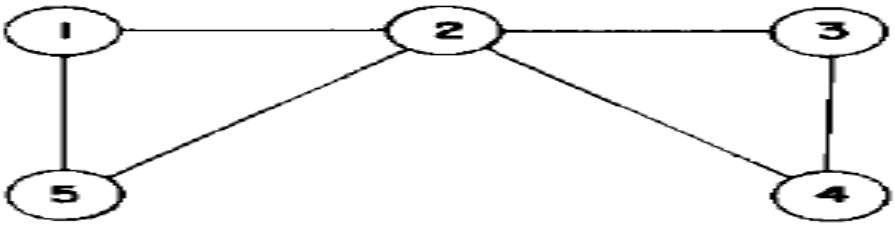

Figure-4.3: Graph

The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
- Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
- The graph may be directed or undirected. Only distinct cycles are output.
- From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}

- The backtracking solution vector (x_1, x_2, --- x_n)
  - x_i i^{th} visited vertex of proposed cycle.
- By using backtracking we need to determine how to compute the set of possible vertices for x_k if x_1,x_2,x_3---x_{k-1} have already been chosen.
  - If k=1 then x1 can be any of the n-vertices.

By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.

This algorithm is started by 1st initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Generating Next Vertex	Finding all Hamiltonian Cycles

Algorithm NextValue(k)
{
// x[1: k-1] is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
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
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
If(j=k) then //if true , then vertex is distinct
If((k<n) or (k=n) and G[x[n], x[1]]≠0))
Then return ;
}
}
Until (false);
}
	Algorithm Hamiltonian(k)
	{
	Repeat{
	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)
	}

Complexity Analysis

In Hamiltonian cycle, in each recursive call one of the remaining vertices is selected in the worst case. In each recursive call the branch factor decreases by 1. Recursion in this case can be thought of as n nested loops where in each loop the number of iterations decreases by one. Hence the time complexity is given by:

$$T(N)=N*(T(N-1)+O(1))$$
$$T(N) = N*(N-1)*(N-2).. = O(N!)$$

GRAPH COLORING :

Let G be a graph and m be a given positive integer. We want to discover 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.

Note that, if 'd' is the degree of the given graph then it can be colored with 'd+1' colors.

The m- colorability optimization problem asks for the smallest integer 'm' for which the graph G can be colored. This integer is referred as "Chromatic number" of the graph.

Example:

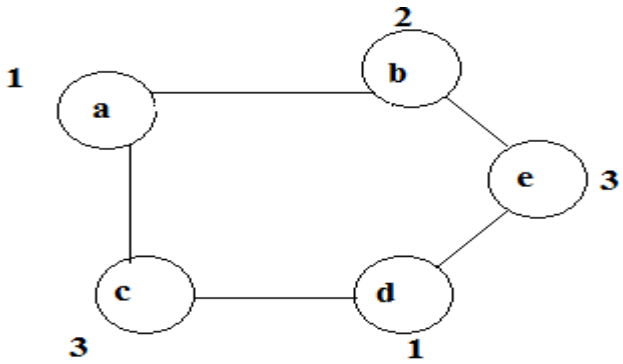

Figure-4.4: Graph Coloring Example

Algorithm:

Finding all m-coloring of a graph	Getting next color

Algorithm mColoring(k){
// g(1:n, 1:n):boolean adjacency matrix.
// k index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m]
	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
	{
	if ((g[k,j]≠0) and (x[k]=x[j]))
	then break;
	}
	if(j=n+1) then return; //new color found
	} until(false)
	}

Complexity Analysis

1) 2-colorability

There is a simple algorithm for determining whether a graph is 2-colorable and assigning colors to its vertices: do a breadth-first search, assigning "red" to the first layer, "blue" to the second layer, "red" to the third layer, etc. Then go over all the edges and check whether the two endpoints of this edge have different colors. This algorithm is $O(|V|+|E|)$ and the last step ensures its correctness.

2) k-colorability for k>2

For k > 2 however, the problem is much more difficult. For those interested in complexity theory, it can be shown that deciding whether a given graph is k-colorable for k > 2 is an NP-complete problem. The first algorithm that can be thought of is brute-force search: consider every possible assignment of k colors to the vertices, and check whether any of them are correct. This of course is very expensive, on the order of $O((n+1)!)$, and impractical. Therefore we have to think of a better algorithm.

INTRODUCTION TO BRANCH & BOUND METHOD:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

- It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
- It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.
- Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node
- Branch and Bound is the generalization of graph search strategies, BFS and D- search.
    - A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
    - A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

EXAMPLES OF BRANCH AND BOUND METHOD

There are lots of problem which can be solve using branch and bound methods. Like:

- **Travelling Salesperson Problem**
- 0/1 knapsack
- Quadratic assignment problem
- Nearest neighbor search

TRAVELLING SALESPERSON PROBLEM

Definition: Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.

A tree of nodes is generated where each node has specified constraints regarding edges connecting two cities in a tour that must be present and edges connecting two cities in a tour that cannot be present. Based on the constraints in a given node, a lower bound is formulated for the given node. This lower bound represents the smallest solution that would be possible if a sub-tree of nodes leading eventually to leaf nodes containing legal tours were generated below the given node. If this lower bound is higher than the best known solution to-date, the node may be pruned. This pruning has the effect of sparing result in a significant saving if the pruned node were relatively near the top of the tree.

Let us explore the mechanism for computing lower bounds for a node.

Example:

	A	B	C	D
A	∞	10	5	3
B	8	∞	9	7
C	1	6	∞	9
D	2	3	8	∞

Now find the reduced matrix by:

- Subtracting the smallest element from row i(for example r1), so one element will become 0 and rest element remain non negative.
- Then after subtracting the smallest element from col j(for example c1), so one element will

become 0 and rest element remain non negative.

- Set element A[I,j] = ∞

So the total reduced cost T= r1 + c1.

Hence the reduced matrix is:

- Subtracted 3 from row 1
- Subtracted 7 from row 2
- Subtracted 1 from row 3
- Subtracted 2 from row 4
- Subtracted 1 from col 2
- Subtracted 2 from col 3

	A	B	C	D
A	∞	6	0	0
B	1	∞	0	0
C	0	4	∞	8
D	0	0	4	∞

Total reduced cost is = 3+7+1+2+1+2=

We examine minimum cost for each node 1,….,4 by using the formula-***l(B)=l(A) + M(i,j) + T***

Here:- l(b) is the cost of new node, l(A) is the cost of previous node, and T is the reduced cost

So the state space tree is given as:

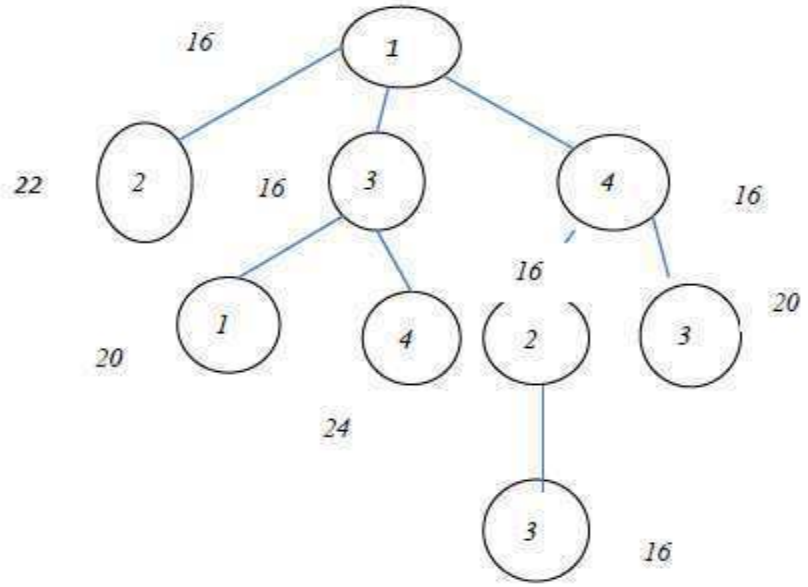

Figure-4.5: State Space Tree for TSP

Here the path is using above state space tree is: 1->4->2->3->1

Algorithm for TSP

function CheckBounds(st,des,cost[n][n])

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) C olumnReduction(reduced)

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

return pencost[des]

end function

function RowMin(cost[n][n],i)

min = cost[i][0]

for j ← 0, n − 1 do

if cost[i][j] < min then

min = cost[i][j]

end if end for return min

end function

function ColMin(cost[n][n],i)

min = cost[0][j]

for i ← 0, n − 1 do

if cost[i][j] < min then

min = cost[i][j]

end if end for return min

end function

function Rowreduction(cost[n][n])

row = 0

for i ← 0, n − 1 do

rmin = rowmin(cost, i)

if rmin /= ∞ then

row = row + rmin

end if

for j ← 0, n − 1 do

if cost[i][j] /= ∞ then

cost[i][j] = cost[i][j] − rmin

end if end for

end for end function

function Columnreduction(cost[n][n])

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] /= ∞ then

cost[i][j] = cost[i][j] − cmin

end if end for

end for end function

function Main

for i ← 0, n − 1 do

select[i] = 0 end for rowreduction(cost) columnreduction(cost) t = row + col

while all visited(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 then

min = edgecost[i]

k = i

end if end if

end for

select[k] = 1

for p ← 1, n − 1 do

cost[j][p] = ∞

end for

for p ← 1, n − 1 do

cost[p][k] = ∞

end for cost[k][j] = ∞ rowreduction(cost) columnreduction(cost)

end while end function

Complexity Analysis:
Traveling salesman problem is a NP-hard problem. Until now, researchers have not found a polynomial time algorithm for traveling salesman problem. Among the existing algorithms, dynamic programming algorithm can solve the problem in time O(n^2*2^n) where n is the number of nodes in the graph. The branch-and-cut algorithm has been applied to solve the problem with a large number of nodes. However, branch-and-cut algorithm also has an exponential worst-case running time.

LOWER AND UPPER BOUND THEORY:
Lower bound is the best case running time. Lower bound is determined by the easiest input. It provides a goal for all input. Whereas upper bound is the worst case and is determined by the most difficult input for a given algorithm. Upper bounds provide guarantees for all inputs i.e. Guarantees on performance of the algorithm when run on different inputs will not perform any worse than over the most difficult input.
There are a number of lower bounds for problems related to sorting, for example element-distinctness: are there two identical elements in a set? These lower bounds are actually interesting because they generalize the comparison-lower bound to more algebraic formulations: for example, you can show that solving element distinctness in a model that allows algebraic operations has a lower bound via analyzing the betti numbers of the space induced by different answers to the problem.
A very interesting example of an unconditional exponential deterministic lower bound (i.e not related to P vs NP) is for estimating the volume of a polytope. There is a construction due to Furedi and Barany that shows that the volume of a convex polytope cannot be approximated to within an even exponential factor in polynomial time unconditionally. This is striking because there are randomized poly-time algorithms that yield arbitrarily good approximations.

• **Lower Bound,** L(n), is a property of the specific problem, i.e. sorting problem, MST, matrix multiplication, not of any particular algorithm solving that problem.
• Lower bound theory says that no algorithm can do the job in fewer than L(n) time units for arbitrary inputs, i.e., that every comparison-based sorting algorithm must take at least L(n) time in the worst case.
• L(n) is the minimum over all possible algorithms, of the maximum complexity.
• **Upper bound** theory says that for any arbitrary inputs, we can always sort in time at most U(n). How long it would take to solve a problem using one of the known Algorithms with worst-case input gives us a upper bound.
• Improving an upper bound means finding an algorithm with better worst-case performance.
• U(n) is the minimum over all known algorithms, of the maximum complexity.
• Both upper and lower bounds are minima over the maximum complexity of inputs of size n.
• The ultimate goal is to make these two functions coincide. When this is done, the optimal algorithm will have L(n) = U(n).

There are few techniques for finding lower bounds

1) Trivial Lower Bounds:

For many problems it is possible to easily observe that a lower bound identical to n exists, where n is the number of inputs (or possibly outputs) to the problem.

• The method consists of simply counting the number of inputs that must be examined and the number of outputs that must be produced, and note that any algorithm must, at least, read its inputs and write its outputs.

2) Information Theory:

The information theory method establishing lower bounds by computing the limitations on information gained by a basic operation and then showing how much information is required before a given problem is solved.

• This is used to show that any possible algorithm for solving a problem must do some minimal amount of work.

• The most useful principle of this kind is that the outcome of a comparison between two items contains one bit of information.

3) Decision Tree Model

• This method can model the execution of any comparison based problem. One tree for each input size *n*.

• View the algorithm as splitting whenever it compares two elements.

• The tree contains the comparisons along all possible instruction traces.

• The running time of the algorithm = the length of the path taken.

• Worst-case running time = the height of tree.

PARALLEL ALGORITHM:

A parallel algorithm can be executed simultaneously on many different processing devices and then combined together to get the correct result. Parallel algorithms are highly useful in processing huge volumes of data in quick time. This tutorial provides an introduction to the design and analysis of parallel algorithms. In addition, it explains the models followed in parallel algorithms, their structures, and implementation.

An algorithm is a sequence of steps that take inputs from the user and after some computation, produces an output. A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

Concurrent Processing

The easy availability of computers along with the growth of Internet has changed the way we store and process data. We are living in a day and age where data is available in abundance. Every day we deal with huge volumes of data that require complex computing and that too, in quick time. Sometimes, we need to fetch data from similar or interrelated events that occur simultaneously. This is where we require concurrent processing that can divide a complex task and process it multiple systems to produce the output in quick time.

Concurrent processing is essential where the task involves processing a huge bulk of complex data. Examples include – accessing large databases, aircraft testing, astronomical calculations, atomic and nuclear physics, biomedical analysis, economic planning, image processing, robotics, weather forecasting, web-based services, etc.

Parallelism is the process of processing several set of instructions simultaneously. It reduces the total computational time. Parallelism can be implemented by using parallel computers, i.e. a computer with many processors. Parallel computers require parallel algorithm, programming languages, compilers and operating system that support multitasking.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in