Module IV

- **Dynamic Programming, Back Tracking and Branch & Bound**
  - **Dynamic Programming Control Abstraction**
    - **The Optimality Principle**
    - **Matrix Chain Multiplication-Analysis**
    - **All Pairs Shortest Path Algorithm - Floyd-Warshall Algorithm-Analysis**
  - **The Control Abstraction of Back Tracking – The N Queen's Problem**
  - **Branch and Bound Algorithm for Travelling Salesman Problem**

- **Dynamic Programming**
  - Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
  - Dynamic Programming is mainly an optimization over plain recursion.
  - Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
  - The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.
  - This simple optimization reduces time complexities from exponential to polynomial.
  - For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.
  - In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.

  - **The Optimality Principle**
    - **Definition**: The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
    - A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themesleves optimal solutions for their subproblems.
    - Examples:
      - The shortest path problem satisfies the Principle of Optimality.
      - This is because if $a,x_1,x_2,...,x_n,b$ is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j.

  - **Characteristics of Dynamic Programming**
    1. **Overlapping Subproblems**
       - Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.
       - Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
       - For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

2. **Optimal Substructure**
   - A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
   - For example, the Shortest Path problem has the following optimal substructure property: If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

- **Steps of Dynamic Programming**
  - Dynamic programming design involves 4 major steps:
    1. Characterize the structure of an optimal solution.
    2. Recursively define the value of an optimal solution.
    3. Compute the value of an optimal solution, typically in a bottom-up fashion.
    4. Construct an optimal solution from computed information.

- **Optimal matrix multiplication**
  - Suppose we wish to compute the product of 4 matrices $A_1 \times A_2 \times A_3 \times A_4$
  - The different parenthesizations are

    $(A_1(A_2(A_3 A_4)))$
    $(A_1((A_2 A_3) A_4))$
    $((A_1A_2)(A_3 A_4))$
    $((A_1(A_2 A_3))A_4)$
    $(((A_1A_2) A_3)A_4)$

  - We can multiply two matrices A and B if and only if they are compatible: The number of columns of A must be equal to the number of rows of B.
  - If A is a pXq matrix and B is a qXr matrix, the resulting matrix C is a pX r matrix.
  - The time to compute C is pqr.
  - We shall express costs in terms of the number of scalar multiplications
  - Example:
    - Consider 3 matrices A_1, A_2 and A_3. Its dimensions are 10X100, 100X5, 5X50 respectively.
    - Number of scalar multiplications for
      - $(A_1A_2) A_3)$ is 7500
      - $(A_1 (A_2A_3))$ is 75000
    - Thus, computing the product according to the first parenthesization is 10 times faster.

  - **Matrix-Chain Multiplication Problem**: Given a chain $(A_1, A_2, \ldots A_n)$ of n matrices, where for i = 1, 2, . . . n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications
  - In the matrix-chain multiplication problem, we are not actually multiplying matrices.
  - Our **goal** is only to determine an order for multiplying matrices that has the lowest cost

- **Matrix Chain Multiplication : Dynamic Programming Method**
  - Step 1: The structure of an optimal parenthesization
    - $A_{i..j}$ denote the matrix that results from evaluating the product $A_iA_{i+1} \ldots A_j$ where i<= j
    - If i< j, we must split the problem into two subproblems ($A_i A_{i+1} \ldots A_k$ and $A_{k+1}A_{i+1} \ldots A_j$), for some integer k in the range i<= k < j.
    - That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$. Then multiply them together to produce the final product $A_{i..j}$.
    - Total cost = Cost of computing the matrix $A_{i..k}$+ Cost of computing $A_{k+1..j}$+ Cost of multiplying them together.

  - Step 2: A recursive solution
    - We can define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.
    - Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
    - For the full problem, the lowest cost way to compute $A_{1..n}$ would thus be m[1, n]
    - $A_{i..i} = A_i$ so m[i,i] = 0 for i=1,2,. .. . n

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

    - s[i,j] be a value of k at which we split the product $A_iA_{i+1} \ldots A_j$ in an optimal parenthesization.
    - This will take exponential time

  - Step 3: Computing the optimal costs
    - Compute the optimal cost by using a tabular, bottom-up approach

Algorithm Matrix_Chain_Order(p)

```
{       n = p.length – 1
        Let m[1..n, 1..n] and s[1..n-1 , 2..n] be new tables
        for i=1 to n do
                m[i,i] = 0
        for l=2 to n do
        {       for i=1 to n-l+1 do
                {       j=i+l-1
                        m[i,j] = α
                        for k=i to j-1 do
                        {       q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
                                if q<m[i,j] then
                                {       m[i,j] = q
                                        s[i,j] = k
                                }
                          }

                }
        }
        return m[][] and s[][]
}
```

- Matrix A_i has dimensions p_{i-1} X p_i for i =1,2, . . . n.
- Input to this algorithm is a sequence p = (p_0, p_1, ... p_n), where p.length = n + 1.
- The procedure uses 2 auxiliary tables
- m[1 .. n , 1...n] for storing the cost of matrix multiplication
- s[1..n-1 , 2 ...n] records which index of k achieved the optimal cost in computing m[i,j] .

- Step 4: Constructing an optimal solution

```
Algorithm Print_Optimal_Parens(s,i,j)
{
        if i==j then
                print "A"i
        else
                print "("
                print_Optimal_Parens(s,i,s[i,j])
                print_Optimal_Parens(s,s[i,j]+1,j)
                print ")"
}
```

- Initial call is PRINT-OPTIMAL-PARENS(s,1,n)

- **Time Complexity**
  - We are generating n(n-1)/2 number of elements in matrix m[].
  - To calculate each element it will take atmost n time.
  - So the time complexity = O(n.n(n-1)/2) = **$O(n^3)$**

- **Examples**
  1. Using Dynamic Programming, find the fully parenthesized matrix product for multiplying the chain of matrices< A1 A2 A3 A4 A5 A6 > whose dimensions are <30X35>, <35X15>, <15X5>, <5X10>, <10X20> and <20X25> respectively
  2. Given a chain of 4 matrices <A1,A2,A3,A4> with dimensions <5X4>,<4X6>,<6X2>,<2X7> respectively. Using Dynamic programming find the minimum number of scalar multiplications needed and also write the optimal multiplication order.
  3. Find an optimal paranthesization of a matrix-chain product whose sequence of dimensions is 4x10,10x3,3x12,12x20,20x7

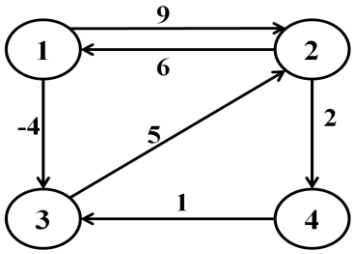
- **All pairs shortest path problem**
  - Find the shortest distances between every pair of vertices in a given weighted directed Graph
  - The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. Negative edge weights are also allowed.
  - As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

- **Floyd Warshall Algorithm**

  Inputs are the adjacency matrix of the given graph and total number of vertices

```
Algorithm FloydWarshall(cost[][], n)
{
        for i=1 to n do
                for j=1 to n do
                        D[i, j] = cost[i, j]
        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
}
```

- **Time Complexity**
  - Floyd Warshall Algorithm consists of three loops over all the nodes. Each loop has constant complexities.
  - Hence, the time complexity of Floyd Warshall algorithm = $O(n^3)$, where n is the number of nodes in the given graph.
- **Example**
  - Consider the following directed weighted graph. Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices



  - Solution
    - Remove all self loops and parallel edges(keeping the lowest weight edge) of the given graph
    - Write the adjacency matrix

$$\mathbf{D^0} = \begin{pmatrix} 0 & 9 & -4 & \alpha \\ 6 & 0 & \alpha & 2 \\ \alpha & 5 & 0 & \alpha \\ \alpha & \alpha & 1 & 0 \end{pmatrix}$$

    - Find the matrix D^1
      - Keep the 1st row, 1st column and diagonal elements of D^0 as such
      - $D^1(2,3) = \min\{ D^0(2,3), D^0(2,1) + D^0(1,3) \} = \min\{\alpha, 6+(-4)) = 2$
      - $D^1(2,4) = \min\{ D^0(2,4), D^0(2,1) + D^0(1,4) \} = \min\{2, 6+\alpha) = 2$
      - $D^1(3,2) = \min\{ D^0(3,2), D^0(3,1) + D^0(1,2) \} = \min\{5, \alpha +9) = 5$
      - $D^1(3,4) = \min\{ D^0(3,4), D^0(3,1) + D^0(1,4) \} = \min\{\alpha, \alpha + \alpha) = \alpha$
      - $D^1(4,2) = \min\{ D^0(4,2), D^0(4,1) + D^0(1,2) \} = \min\{\alpha, \alpha +9) = \alpha$
      - $D^1(4,3) = \min\{ D^0(4,3), D^0(4,1) + D^0(1,3) \} = \min\{1, \alpha +(-4)) = 1$

$$\mathbf{D^1} = \begin{pmatrix} 0 & 9 & -4 & \alpha \\ 6 & 0 & 2 & 2 \\ \alpha & 5 & 0 & \alpha \\ \alpha & \alpha & 1 & 0 \end{pmatrix}$$

- Find the matrix D^2
  - Keep the 2nd row, 2nd column and diagonal elements of D^1 as such
  - $D^2(1,3) = \min\{ D^1(1,3), D^1(1,2) + D^1(2,3) \} = \min\{-4, 9+2) = -4$
  - $D^2(1,4) = \min\{ D^1(1,4), D^1(1,2) + D^1(2,4) \} = \min\{\alpha, 9+2) = 11$
  - $D^2(3,1) = \min\{ D^1(3,1), D^1(3,2) + D^1(2,1) \} = \min\{\alpha, 5+6) = 11$
  - $D^2(3,4) = \min\{ D^1(3,4), D^1(3,2) + D^1(2,4) \} = \min\{\alpha, 5+2) = 7$
  - $D^2(4,1) = \min\{ D^1(4,1), D^1(4,2) + D^1(2,1) \} = \min\{\alpha, \alpha+6) = \alpha$
  - $D^2(4,3) = \min\{ D^1(4,3), D^1(4,2) + D^1(2,3) \} = \min\{1, \alpha+2) = 1$

$$\mathbf{D^2} = \begin{pmatrix} 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \alpha & \alpha & 1 & 0 \end{pmatrix}$$

- Find the matrix D^3
  - Keep the 3rd row, 3rd column and diagonal elements of D^2 as such
  - $D^3(1,2) = \min\{ D^2(1,2), D^2(1,3) + D^2(3,2) \} = \min\{9, -4+5) = 1$
  - $D^3(1,4) = \min\{ D^2(1,4), D^2(1,3) + D^2(3,4) \} = \min\{11, -4+7) = 3$
  - $D^3(2,1) = \min\{ D^2(2,1), D^2(2,3) + D^2(3,1) \} = \min\{6, 2+11) = 6$
  - $D^3(2,4) = \min\{ D^2(2,4), D^2(2,3) + D^2(3,4) \} = \min\{2, 2+7) = 2$
  - $D^3(4,1) = \min\{ D^2(4,1), D^2(4,3) + D^2(3,1) \} = \min\{\alpha, 1+11) = 12$
  - $D^3(4,2) = \min\{ D^2(4,2), D^2(4,3) + D^2(3,2) \} = \min\{\alpha, 1+5) = 6$

$$\mathbf{D^3} = \begin{pmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{pmatrix}$$

- Find the matrix D^4
  - Keep the 4th row, 4th column and diagonal elements of D^3 as such
  - $D^4(1,2) = \min\{ D^3(1,2), D^3(1,4) + D^3(4,2) \} = \min\{1, 3+6) = 1$
  - $D^4(1,3) = \min\{ D^3(1,3), D^3(1,4) + D^3(4,3) \} = \min\{-4, 3+1) = -4$
  - $D^4(2,1) = \min\{ D^3(2,1), D^3(2,4) + D^3(4,1) \} = \min\{6, 2+12) = 6$
  - $D^4(2,3) = \min\{ D^3(2,3), D^3(2,4) + D^3(4,3) \} = \min\{2, 2+1) = 2$
  - $D^4(3,1) = \min\{ D^3(3,1), D^3(3,4) + D^3(4,1) \} = \min\{11, 7+12) = 11$
  - $D^4(3,2) = \min\{ D^3(3,2), D^3(3,4) + D^3(4,2) \} = \min\{5, 7+6) = 5$

$$\mathbf{D^4} = \begin{pmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{pmatrix}$$

- D^4 represents the shortest distance between each pair of the given graph

- **Comparison of Divide and Conquer and Dynamic Programming strategies**
  - Both techniques split their input into parts, find sub-solutions to the parts, and synthesize larger solutions from smaller ones.
  - Divide and Conquer splits its input at pre-specified deterministic points (e.g., most probably in the middle)
  - Dynamic Programming splits its input at every possible split points rather than at pre-specified points. After trying all split points, it determines which split point is optimal.
  - Divide & Conquer algorithm partition the problem into **disjoint subproblems**. Solve the subproblems recursively and then combine their solution to solve the original problems.
  - Dynamic Programming is used when the subproblems are **not independent**, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.
  - Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

- **Greedy vs. Dynamic Programming**
  - Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
  - The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
  - Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
  - There is no a priori litmus test by which one can tell if the Greedy method will lead to an optimal solution.
  - By contrast, there is a litmus test for Dynamic Programming, called The Principle of Optimality
  - The greedy method only generated one decision sequence ever.
  - In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal and so will not generated.

- **Back Tracking**
  - Backtracking method expressed the solution as n-tuple (x_1, x_2.,…… x_n), where x_i's are chosen from some finite set S_i.
  - The problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a criterion function P(x_1, x_2.,…… x_n).
  - Examples: Sorting the array of integers in a[l: n]
    - The solution to the problem is expressed an n-tuple, where x_i is the index of the i^{th} smallest element.
    - The criterion function P is: $a[x_i] \leq a[x_{i+1}]$, for $1 \leq i < n$.
    - The set $S_i = \{1,2, \ldots .n\}$
    - Different methods for solving this problem
      - **Brute Force approach**
        - Suppose m_i is the size of set S_i.
        - The number of tuples (with size n) that are possible candidates for satisfying the function P is: $m = m_1 \times m_2 \times m_3 \ldots \ldots \times m_n$
        - Brute Force approach evaluates each one with P, and save those which yield the optimum.

- **Backtracking algorithm**
  - It yields the same answer with far fewer than m trials.
  - Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions (bounding functions) $P_i(x_1, x_2.,\ldots\ldots x_i)$ to test whether the vector being formed has any chance of success.
  - The major advantage of this method is that if it is realized that the partial vector $(x_1, x_2.,\ldots\ldots x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \times m_{i+2}. \ldots . \times m_n$ possible test vectors can be ignored entirely.

- Backtracking method require that all the solutions satisfy a complex set of constraints. These constraints can be divided into two categories:

  - **Explicit Constraints**
    - Explicit constraints are rules that restrict each x_i to take on values only from a given set
    - The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible **solution space** for I.
    - Example:

$$\begin{array}{rcl} x_i \geq 0 & \text{or} \quad S_i & = \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} \quad S_i & = \{0,1\} \\ l_i \leq x_i \leq u_i & \text{or} \quad S_i & = \{a : l_i \leq a \leq u_i\} \end{array}$$

  - **Implicit Constraints**
    - These are rules that determine which of the tuples in the solution space of I satisfy the criterion function (Bounding Function).

- **N-Queens Problem**
  - n queens are to be placed on a n x n chessboard so that no two attack. That is, no two queens are on the same row, column, or diagonal.
  - Number the rows and columns of the chessboard 1through n.
  - The queens can also be numbered 1through n.
  - Since each queen must be on a different row, we can assume that queen i is to be placed on row i.
  - All **solutions** to the n-queens problem can therefore be represented as n-tuples$(x_1, x_2.,\ldots\ldots x_n)$, where x_i is the column on which queen i is placed.
  - **Explicit constraint**: $S_i = \{1,2,3, \ldots . n\}$, $1 \leq i \leq n$
    - The solution space contains $|S_1| \times |S_2| \times \ldots\ldots \times |S_n| = n^n$ tuples.
  - **Implicit constraints**:
    - No two x_i's can be the same(i.e. all queens must be on different columns)
      - The solution space contains $|S_1| \times |S_2| \times \ldots\ldots \times |S_n| = n(n-1) \ldots .1 = n!$ tuples
      - It reduces the solution space from n^n to n!.
    - No two queens can be on the same diagonal.

	1	2	3	4
1		**Q1**		
2				**Q2**
3	**Q3**			
4			**Q4**	

- Following is a tree organization (**permutation tree/State Space Tree**) for 4-queen problem without considering the last implicit constraint.

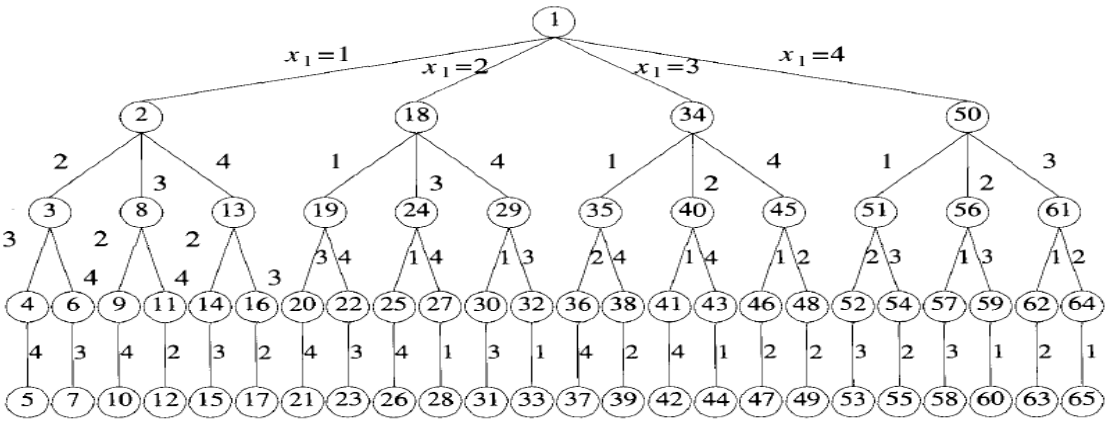
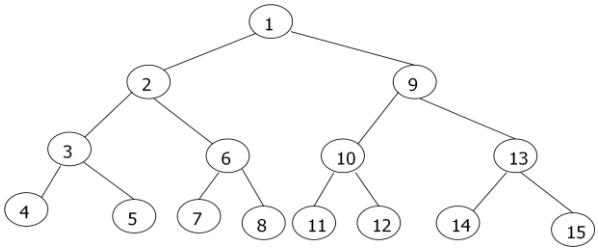


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

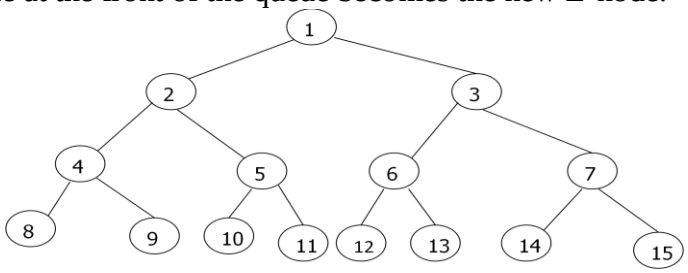
- The edges are labeled by possible values of x_i.
- Edges from level 1 to level 2 nodes specify the values for x_1
- Edges from level i to level i +1 are labeled with the values of x_i
- Thus, the leftmost sub-tree contains all solutions with $x_1 = 1$; its leftmost sub-tree contains all solutions with $x_1 = 1$ and $x_2 = 2$, and soon.
- The **solution space** is defined by all paths from the root node to a leaf node. There are 4!= 24 leaf nodes in the above tree.

- **Permutation tree** or **State Space Tree**
  - Tree organization of solution space is called state space tree.
  - **Problem State**: Each node in this tree defines a problem state.
  - **State Space of the problem**: All paths from the root to other nodes define the State Space of the problem.
  - **Solution States**: Those problem states s for which the path from the root to s defines a tuple in the solution space.
  - **Answer states**: Those solution states s for which the path from the root to s defines a tuple that satisfies all implicit constraints of the problem.
  - **Live Node**: A node which has been generated and all of whose children have not yet been generated is called a live node.
  - **E-node** : The live node whose children are currently being generated is called the E-node
  - **Dead node**: It is a generated node which is not to be expanded further or all of whose children have been generated.
  - **Bounding functions** are used to kill live nodes without generating all their children.
  - Problem States can be generated by:
    - **Depth First generation of the problem states**:
      - As soon as a new child C of the current E-node R is generated, this child will become the new E-node.
      - Then R will become the E-node again when the sub-tree C has been fully explored.

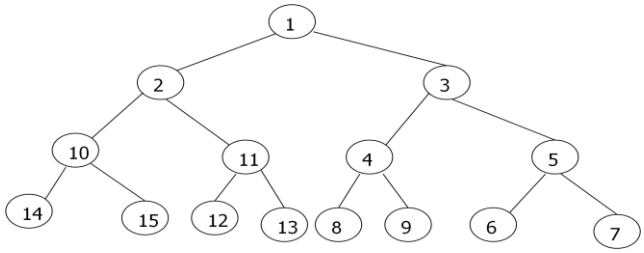


- **Backtracking**: Depth first node generation with bounding functions.

- **Breadth First generation of the problem states**:
  - The -E-node remains the E-node until it is dead.
  - **Branch-and-bound methods**: Breadth first node generation method with bounding function is called Branch and Bound method. There are two alternatives
    - **Breadth First Generation Method**: Each new node is placed into a queue. When all the children of the current-E-node have been generated, the next node at the front of the queue becomes the new E-node.



    - **D-search(depth search):** Each new node is placed into a stack. When all the children of the current-E-node have been generated, the next node at the top of the stack becomes the new E-node.



  - At the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions.

- **Backtracking works on 4-Queens Problem**
  - If (x_1, x_2.,…… x_i) is the path to the current E-node, then all children nodes with parent-child labeling x_{i+1} are such that (x_1, x_2.,…… x_{i+1}) represents a chessboard configuration in which no two queens are attacking.
  - We start with the root node as the only live node. This becomes the E-node and the path is (). We generate one child(node 2) and the path is now (1). This corresponds to placing queen1on column1.
  - Node2 becomes the E-node. Node3 is generated and immediately killed.
  - The next node generated is node8 and the path becomes(1,3). Node8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node.
  - We back track to node2 and generate another child node13.The path is now (1,4). This process continues until it will generate a proper arrangement.

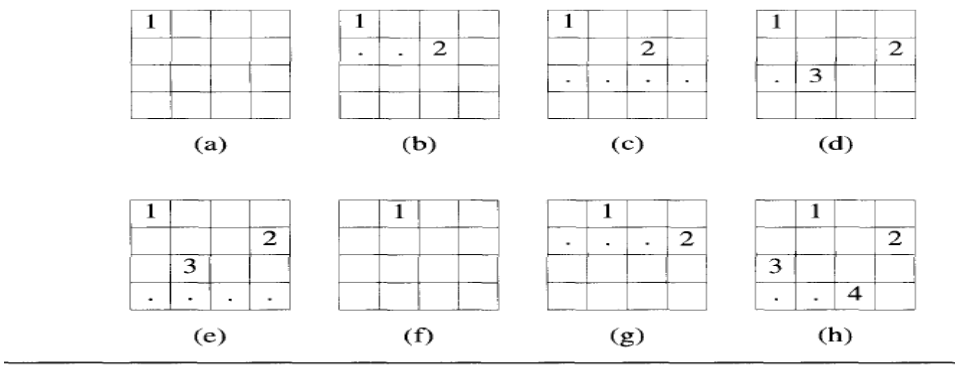
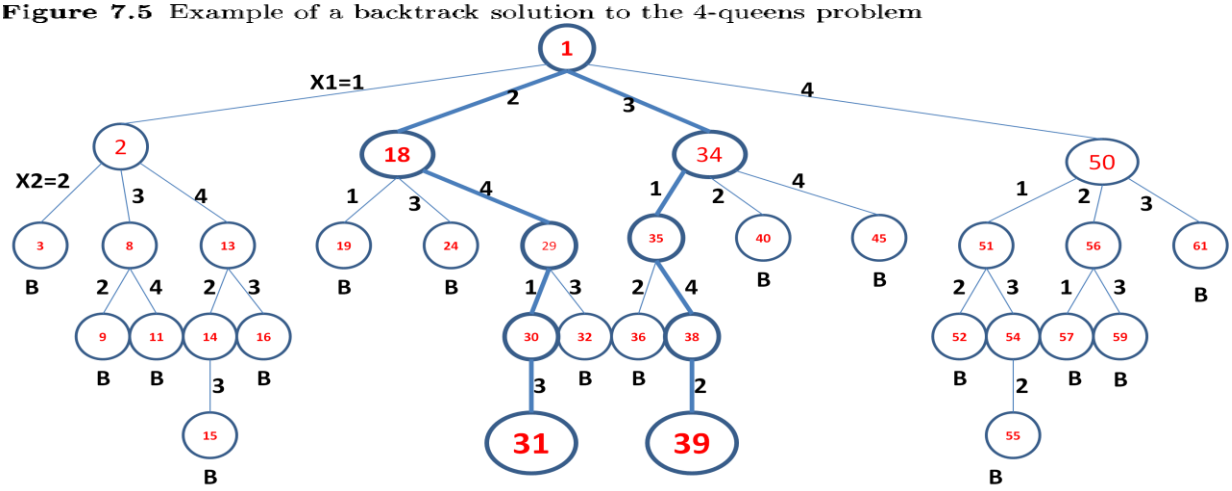


Figure 7.5 Example of a backtrack solution to the 4-queens problem



State Space Tree of 4 Queens Problem

- **Backtracking Control Abstraction**
  - (x_1, x_2.,…… x_i) be a path from the root to a node in a state space tree.
  - **Generating Function** T(x_1, x_2.,…… x_i) be the set of all possible values for x_{i+1} such that (x_1, x_2.,…… x_{i+1}) is also a path to a problem state.
  - T(x_1, x_2.,…… x_n) = φ
  - **Bounding function** B_{i+1}(x_1, x_2.,…… x_{i+1}) is false for a path (x_1, x_2.,…… x_{i+1}) from the root node to a problem state, then the path cannot be extended to reach an answer node.
  - Thus the candidates for position i+1of the solution vector (x_1, x_2.,…… x_n) are those values which are generated by T and satisfy B_{i+1}.
  - The recursive version is initially invoked by **Backtrack(1)**.

Backtracking Control Abstraction

```
Algorithm Backtrack(k)
{
        for (each x[k] ϵ T(x[1], . . . .x[k-1])
        {
            if(Bk(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)
            }
        }
}
```

- All the possible elements for the k^{th} position of the tuple that satisfy B_k are generated one by one, and adjoined to the current vector (x_1, x_2.,…… x_{k-1}).
- Each time x_k is attached, a check is made to determine whether a solution has been found. Then the algorithm is recursively invoked.
- When the for loop is exited, no more values for x_k exist and the current copy of Backtrack ends. The last unresolved call now resumes.
- This algorithm causes all solutions to be printed. If only a single solution is desired, then a flag can be added as a parameter to indicate the first occurrence of success.

- **N-Queens Problem(Cond…)**
  - Consider an n x n chessboard and try to find all possible way to place n non-attacking queens.
  - (x_1, x_2.,…… x_n) be the solution vector. Queen i is placed in i^{th} row and x_i^{th} column. x_i will all be distinct since no two queens can be placed in the same column.
  - There are 2 type of diagonals
    - Positive Diagonal
      - Diagonal from upper left to lower right
      - Every element on the same diagonal has the same **row-column** value
      - Suppose 2 queens are place at position (i,j) and (k,l), then **i-j = k-l**
    - Negative Diagonal
      - Diagonal from upper right to lower left
      - Every element on the same diagonal has the same **row+column** value
      - Suppose 2 queens are place at position (i,j) and (k,l), then **i+j = k+l**
    - The 1^{st} equation implies: i-k = j-l
    - The 2^{nd} equation implies: i-k = l - j
    - Combining these two, we will get : | i-k | = | j-l |
      Absolute value of column difference is equal to the absolute value of row difference.

```
Algorithm NQueens(k,n)
{
        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)
                }
        }
}
```

```
Algorithm Place(k,i)
{
        for j=1 to k-1 do
        {
                if( (x[j]==i) or ( Abs(j-k)==Abs(x[j]-i) )  then
                        Return false
        }
        Return true
}
```

- **Place(k,i)** returns true if the k^{th} queen can be placed in column i.
  - i should be distinct from all previous values x[1],x[2], ..., x[k-1]
  - And no 2 queens are to be on the same diagonal
  - **Time complexity** of Place() is **O(k)**
- **NQueen()** is initially invoked by NQueen(1,n)
  - Time complexity of NQueen() is **O(n!)**

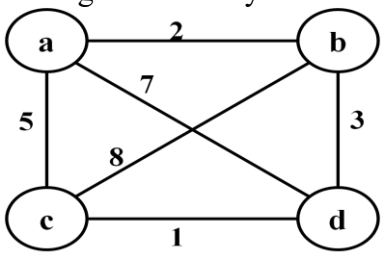
- **Examples**
  - Show the state space tree for 4 Queens problem. Show the steps in solving 4 Queens problem using backtracking method to print all the solutions

- **Branch and Bound Technique**
  - During state space tree generation E-node remains E-node until it is dead.
  - Two strategies:
    - Breadth First Search(BFS)
      - It is called FIFO(First In First Out). Here the live nodes are placed in a queue.
    - Depth Search(D-Search)
      - It is called LIFO(Last In First Out). Here the live nodes are placed in a stack.
  - Least Cost Search(LC Search)
    - To improve the searching speed, we can use a **ranking function ĉ(.)** for live nodes.
    - ĉ(.) value of each live node is calculated. The next E-node is the live node with least ĉ(.). Such a search strategy is called LC Search**.**
    - BFS and D-Search are the special cases of LC-Search.
    - LC-Search coupled with bounding function is called LC Branch and Bound Search.
  - **LC-Search Control Abstraction**

```
Algorithm LCSearch(t)
{       if t is an answer node then output t     and return
        E = t
        Initialize the list of live nodes to be empty
        repeat
        {       for each child x of E do
                {
                  if x is an answer node then output the path from x to t and return
                  Add(x)
                  x → parent = E
                }
                if there are no more live nodes then
                {       Write "no answer node"
                        return
                }
                E = Least()
        }until(false)
}
```

- Least() finds a live node with least ĉ. This node is deleted from the list of live nodes and returned.
- Add(x) adds the new live node x to the list of live nodes.
- LCSearch outputs the path from the answer node to the root t.
- LCSearch terminates only when either an answer node is found or the entire state space tree has been generated and searched.
- The control abstraction for LC, FIFO and LIFO are same. The only difference is the implementation of the list of live nodes.
  - FIFO Search scheme:
    - The list of live nodes is implemented as queue.
    - Least() and Add(x) being algorithms to delete an element from and add an element to the queue.
  - LIFO Search scheme:
    - The list of live nodes is implemented as stack.
    - Least() and Add(x) being algorithms to delete an element from and add an element to the stack.
  - LC-Search Scheme:
    - Add(x) is an algorithm to add elements to the list. Least() returns a live node with least ĉ(.) from the list.

- **Branch and Bound Algorithm for Travelling Salesman Problem**
  - Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.
  - **Example**: Apply branch and bound algorithm to solve TSP for the following graph, assuming the start city as 'a'. Draw the state space tree.



  - Solution
    - The adjacency matrix is

	a	b	c	d
a	α	2	5	7
b	2	α	8	3
c	5	8	α	1
d	7	3	1	α

    - Perform row reduction, then column reduction

α	2	5	7	-2	
2	α	8	3	-2	
5	8	α	1	-1	
7	3	1	α	-1	

Row reduction →

α	0	3	5
0	α	6	1
4	7	α	0
6	2	0	α
0	0	0	0

Column reduction →

α	0	3	5
0	α	6	1
4	7	α	0
6	2	0	α

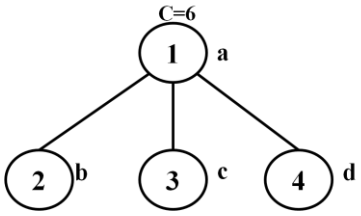
M_1

Total cost reduced = 2+2+1+1+0+0+0+0 = 6

The state space tree is

M_1 is the matrix for node 1 is

- Generate the child node of node 1



- Find the matrix and cost of node 2
  - Set row a and column b elements are α
  - Set $M_1[b, a]= \alpha$
  - The resultant matrix is

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 6 & 1 \\ 4 & \alpha & \alpha & 0 \\ 6 & \alpha & 0 & \alpha \end{pmatrix}$$

  - Perform row reduction, then column reduction

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 6 & 1 \\ 4 & \alpha & \alpha & 0 \\ 6 & \alpha & 0 & \alpha \end{pmatrix} \begin{matrix} 0 \\ -1 \\ 0 \\ 0 \end{matrix} \xrightarrow{\text{Row reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 5 & 0 \\ 4 & \alpha & \alpha & 0 \\ 6 & \alpha & 0 & \alpha \end{pmatrix} \xrightarrow{\text{Column reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 5 & 0 \\ 0 & \alpha & \alpha & 0 \\ 2 & \alpha & 0 & \alpha \end{pmatrix} = M_2$$

$$\begin{matrix} & & & \\ & & & \\ & & & \\ -4 & 0 & 0 & 0 \end{matrix}$$

Cost reduced = r = 5

M_2 is the matrix for node 2

Cost of node 2 = Cost of node 1 + $M_1[a, b]$ + r = 6 + 0 + 5 = 11

- Find the matrix and cost of node 3
  - Set row a and column c elements are α
  - Set $M_1[c, a]= \alpha$
  - The resultant matrix is

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 1 \\ \alpha & 7 & \alpha & 0 \\ 6 & 2 & \alpha & \alpha \end{pmatrix}$$

  - Perform row reduction, then column reduction

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 1 \\ \alpha & 7 & \alpha & 0 \\ 6 & 2 & \alpha & \alpha \end{pmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ -2 \end{matrix} \xrightarrow{\text{Row reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 1 \\ \alpha & 7 & \alpha & 0 \\ 4 & 0 & \alpha & \alpha \end{pmatrix} \xrightarrow{\text{Column reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 1 \\ \alpha & 7 & \alpha & 0 \\ 4 & 0 & \alpha & \alpha \end{pmatrix} = M_3$$

$$\begin{matrix} & & & \\ & & & \\ & & & \\ 0 & 0 & 0 & 0 \end{matrix}$$

Cost reduced = r = 2

M_3 is the matrix for node 3

Cost of node 3 = Cost of node 1 + M_1[a, c] + r = 6 + 3 + 2 = 11

- Find the matrix and cost of node 4
  - Set row a and column d elements are α
  - Set M_1[d, a]= α
  - The resultant matrix is

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & 6 & \alpha \\ 4 & 7 & \alpha & \alpha \\ \alpha & 2 & 0 & \alpha \end{pmatrix}$$

  - Perform row reduction, then column reduction

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & 6 & \alpha \\ 4 & 7 & \alpha & \alpha \\ \alpha & 2 & 0 & \alpha \end{pmatrix} \begin{matrix} 0 \\ 0 \\ -4 \\ 0 \end{matrix} \xrightarrow{\text{Row reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & 6 & \alpha \\ 0 & 3 & \alpha & \alpha \\ \alpha & 2 & 0 & \alpha \end{pmatrix} \xrightarrow{\text{Column reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & 6 & \alpha \\ 0 & 1 & \alpha & \alpha \\ \alpha & 0 & 0 & \alpha \end{pmatrix} = M_4$$

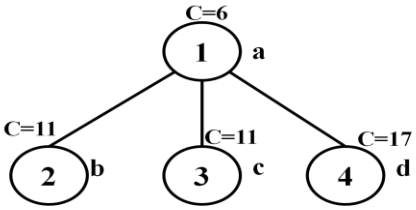
$$\begin{matrix} 0 & -2 & 0 & 0 \end{matrix}$$

Cost reduced = r = 6

M_4 is the matrix for node 4

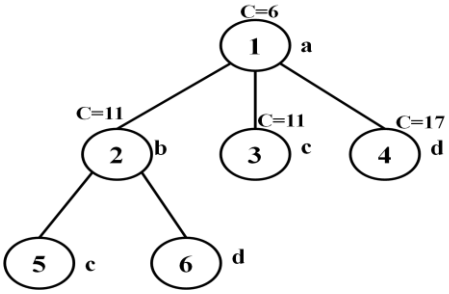
Cost of node 4 = Cost of node 1 + M_1[a, d] + r = 6 + 5 + 6 = 17

  - Now the state space tree is



Now the live nodes are 2, 3 and 4. Minimum cost is for node 2 and 3. Choose one node(say node 2) as the next E-node.

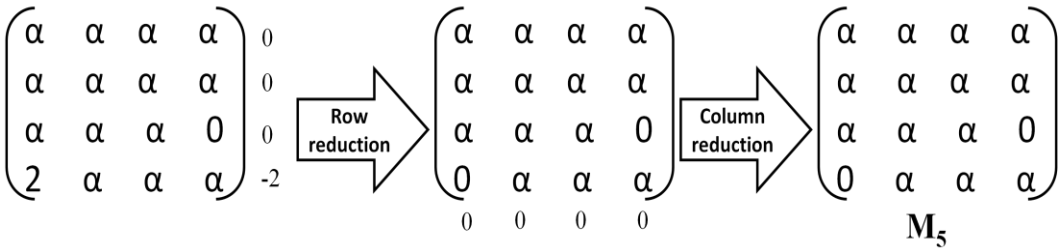
  - Generate the child node of node 2



- Find the matrix and cost of node 5
  - Set row b and column c elements are α
  - Set M_2[c, a]= α
  - The resultant matrix is

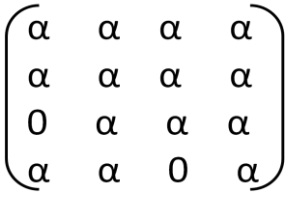
$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & 0 \\ 2 & \alpha & \alpha & \alpha \end{pmatrix}$$

- Perform row reduction, then column reduction

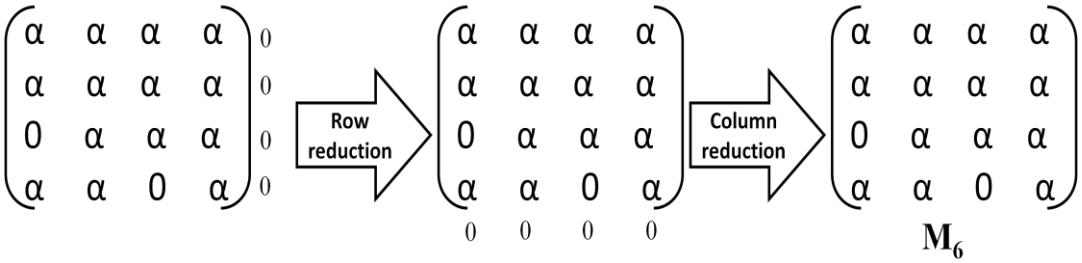


Cost reduced = r = 2
M_5 is the matrix for node 5
Cost of node 5 = Cost of node 2 + M_1[b, c] + r = 11 + 5 + 2 = 18

- Find the matrix and cost of node 6
  - Set row b and column d elements are α
  - Set M_2[d, a]= α
  - The resultant matrix is

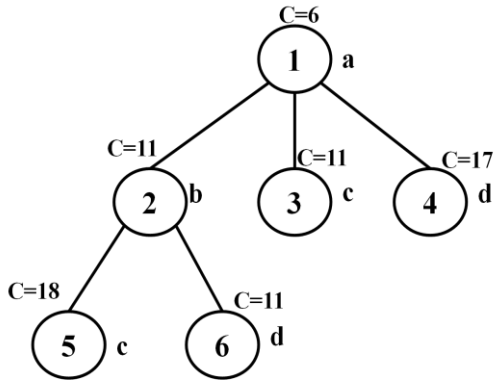


  - Perform row reduction, then column reduction



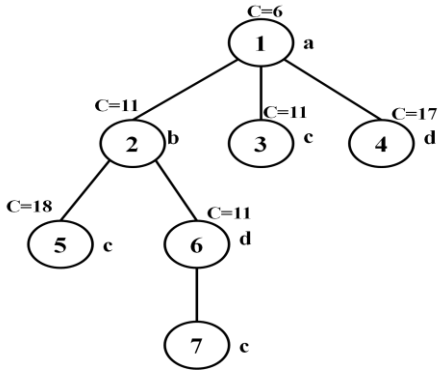
Cost reduced = r = 0
M_6 is the matrix for node 6
Cost of node 6 = Cost of node 2 + M_1[b, d] + r = 11 + 0 + 0 = 11

  - Now the state space tree is



Now the live nodes are 5, 6, 3 and 4. Choose one node which having minimum cost(say node 6) as the next E-node.

- Generate the child node of node 6



- Find the matrix and cost of node 7
  - Set row d and column c elements are α
  - Set $M_6[c, a] = \alpha$
  - The resultant matrix is

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \end{pmatrix}$$

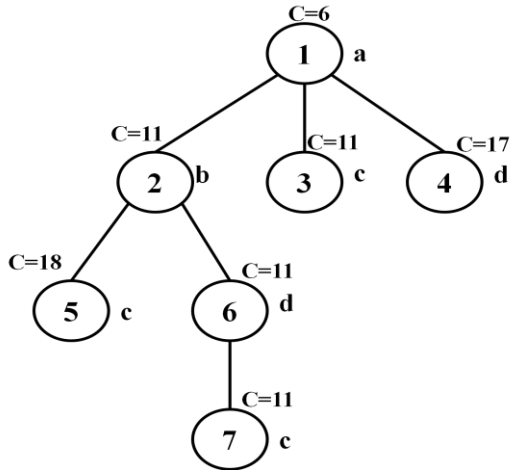
  - Perform row reduction, then column reduction

$$\begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \end{pmatrix} \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \end{matrix} \xrightarrow{\text{Row reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \end{pmatrix} \xrightarrow{\text{Column reduction}} \begin{pmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \end{pmatrix} = M_7$$

$$\begin{matrix} 0 & 0 & 0 & 0 \end{matrix}$$

Cost reduced = r = 0
M_7 is the matrix for node 7
Cost of node 7 = Cost of node 6 + $M_6[d, c]$ + r = 11 + 0 + 0 = 11

  - Now the state space tree is



Now the live nodes are 5, 7, 3 and 4. Choose one node which having minimum cost(say node 7) as the next E-node. Node 7 having no child node.
Now we can say that node 1 to node 7 path is the Traveling salesperson path.

The TSP path = a-b-d-c-a
The TSP cost = 11