Dynamic Programming

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of sub problems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time. Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, 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.

Optimal Sub-Structure

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**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four 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 the computed information.

Difference between Dynamic programming and Divide and Conquer Technique

Divide & Conquer algorithm partition the problem into disjoint sub problems solve the sub problems recursively and then combine their solution to solve the original problems. Divide & Conquer algorithm can be a **Bottom-up approach** and **Top down approach.**

Dynamic Programming is used when the sub problems are not independent, e.g. when they share the same sub problems. 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 sub problems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Divide & Conquer Method	Dynamic Programming
It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of sub problems. **Conquer** the sub problems by solving them recursively. **Combine** the solution to the sub problems into the solution for original sub problems.	It involves the sequence of four steps: o Characterize the structure of optimal solutions. o Recursively defines the values of optimal solutions. o Compute the value of optimal solutions in a Bottom-up minimum. o Construct an Optimal Solution from computed information.
It is Recursive.	It is non Recursive.
It does more work on sub problems and hence has more time consumption.	It solves sub problems only once and then stores in the table.
It is a top-down approach.	It is a Bottom-up approach.
In this sub problems are independent of each other.	In this sub problems are interdependent.
For example: Merge Sort & Binary Search etc.	**For example:** Matrix Multiplication.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure:** Decompose the given problem into smaller sub problems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because sub problem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller sub problems to solve larger sub problems and eventually arrives at a solution to complete problem.

Components of Dynamic programming

1. Stages: The problem can be divided into several sub problems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. States: Each stage has several states associated with it. The states for the shortest path problem were the node reached.
3. Decision: At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. Optimal policy: It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states do not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exists a recursive relationship that identifies the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Applications of dynamic programming

1. **0/1 knapsack problem**
2. **All pair Shortest path problem**
3. Reliability design problem
4. **Longest common subsequence (LCS)**
5. Flight control and robotics control
6. Time-sharing: It schedules the job to maximize CPU usage

Example:

Longest Common Sequence (LCS)

- A subsequence of a given sequence is just the given sequence with some elements left out.
- Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.
- In the longest common subsequence problem, we are given two sequences X = (x1 x2....xm) and Y = (y1 y2 yn) and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Characteristics of Longest Common Sequence

A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given a sequence X = (x1 x2.....xm) we define the ith prefix of X for i=0, 1, and 2...m as Xi= (x1 x2.....xi). For example: if X = (A, B, C, B, C, A, B, C) then X4= (A, B, C, B)

Optimal Substructure of an LCS: Let X = (x1 x2....xm) and Y = (y1 y2.....) yn) be the sequences and let Z = (z1 z2......zk) be any LCS of X and Y.

- If xm = yn, then zk=x_m=yn and Zk-1 is an LCS of Xm-1and Yn-1
- If xm ≠ yn, then zk≠ xm implies that Z is an LCS of Xm-1and Y.
- If xm ≠ yn, then zk≠yn implies that Z is an LCS of X and Yn-1

Step 2: Recursive Solution: LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of Xm-1 and Yn-1. If xm ≠ yn, then we must solve two subproblems finding an LCS of X and Yn-1.Whenever of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of Xm-1 and Yn-1.

Let c [i,j] be the length of LCS of the sequence Xiand Yj.If either i=0 and j =0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem given the recurrence formula

Step3: Computing the length of an LCS: let two sequences X = (x1 x2.....xm) and Y = (y1 y2..... yn) as inputs. It stores the c [i,j] values in the table c [0......m,0..........n].Table b [1..........m, 1..........n] is maintained which help us to construct an optimal solution. c [m, n] contains the length of an LCS of X,Y.

Algorithm of Longest Common Sequence

```
LCS-LENGTH (X, Y)
 1. m ← length [X]
 2. n ← length [Y]
 3. for i ← 1 to m
 4. do c [i,0] ← 0
 5. for j ← 0 to m
 6. do c [0,j] ← 0
 7. for i ← 1 to m
 8. do for j ← 1 to n
 9. do if xi= yj
 10. then c [i,j] ← c [i-1,j-1] + 1
 11. b [i,j] ← "↖"
 12. else if c[i-1,j] ≥ c[i,j-1]
 13. then c [i,j] ← c [i-1,j]
 14. b [i,j] ← "↑"
 15. else c [i,j] ← c [i,j-1]
 16. b [i,j] ← "← "
 17. return c and b.
```

0/1 Knapsack Problem: Dynamic Programming Approach:

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth vi dollars and weight wi pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- vi wi W are integers.

1. W ≤ capacity
2. Value ← Max

Example

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (pi/wi), if we apply dynamic approach to solve this problem, first item A will be selected as it will contribute maximum profit among all the elements.

After selecting item A, no more item will be selected. Hence, for this given set of items total profit is 24. Whereas, the optimal solution can be achieved by selecting items, B and C, where the total profit is 18 + 18 = 36.

Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If A = [aij] is a p x q matrix

B = [bij] is a q x r matrix

C = [cij] is a p x r matrix

Then $AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj}$

Given following matrices {A1, A2, A3,...An} and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

A1 xA2 x, A3 x.....x An

Matrix Multiplication operation is associative in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to parenthesize the above multiplication depending upon our need.

In general, for 1≤ i≤ p and 1≤ j ≤ r

$$C\,[i, j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension p x r Also each entry takes O (q) times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension p q r.

It is also noticed that we can save the number of operations by reordering the parenthesis.

Example:

Let us have 3 matrices, A1,A2,A3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. A1,(A2,A3): First multiplying(A2 and A3) then multiplying and resultant withA1.
2. (A1,A2),A3: First multiplying(A1 and A2) then multiplying and resultant withA3.

No of Scalar multiplication in Case 1 will be:

1. (100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

1. (100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required.

Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

(A1) (A2,A3,A4,................An)

Or (A1,A2) (A3,A4 ....An)

Or (A1,A2,A3) (A4 ....An)

....

Or(A1,A2,A3.............An-1) (An)

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

$$p(n) = \begin{Bmatrix} 1 & \text{if} \quad n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if} \quad n \geq 2 \end{Bmatrix}$$

Also p (n) = c (n-1) where c (n) is the nth Catalon number

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

On applying Stirling's formula we have

$$c(n) = \Omega \left(\frac{4^n}{n^{1.5}}\right)$$

Which shows that 4n grows faster, as it is an exponential function, then n1.5.

DAA - Travelling Salesman Problem

Problem Statement

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

Solution

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

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

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

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

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

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

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

$C(S,j) = \min C(S-\{j\},i)+d(i,j)$

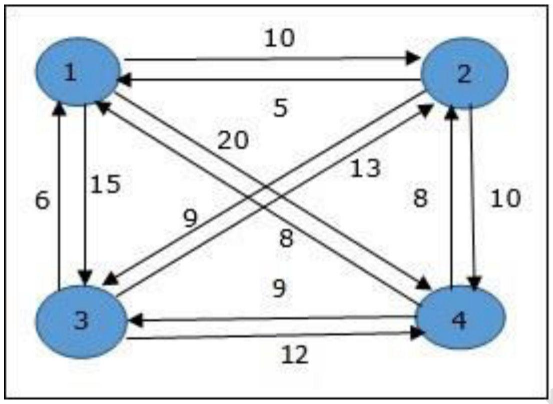
where $i \in S$ and $i \neq j$

c(S,j) = min C(s−{j},i)+d(i,j)

where i∈ S and i≠j

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1) = d(2,1) = 5

Cost(3,Φ,1) = d(3,1) = 6

Cost(4,Φ,1) = d(4,1) = 8

S = 1

Cost(i,s) = min{Cost(j,s–(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1) = d[2,3]+Cost(3,{4},1)=9+20=29

d[2,4]+Cost(4,{3},1)=10+15=25=25

Cost(3,{2,4},1) = d[3,2]+Cost(2,{4},1)=13+18=31

d[3,4]+Cost(4,{2},1)=12+13=25=25

Cost(4,{2,3},1) = d[4,2]+Cost(2,{3},1)=8+15=23

d[4,3]+Cost(3,{2},1)=9+18=27=23

S = 3

Cost(1,{2,3,4},1) = d[1,2]+Cost(2,{3,4},1)=10+25=35

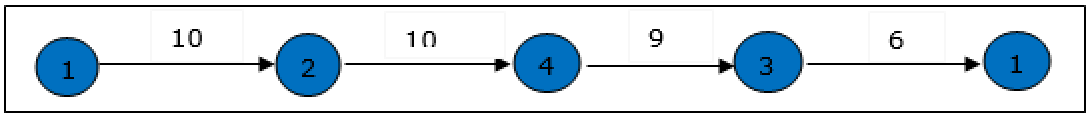
d[1,3]+Cost(3,{2,4},1)=15+25=40

d[1,4]+Cost(4,{2,3},1)=20+23=43=35

The minimum cost path is 35.

Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2]. When s = 3, select the path from 1 to 2 (cost is 10) then go backwards. When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.

When s = 1, we get the minimum value for d [4, 3]. Selecting path 4 to 3 (cost is 9), then we shall go to then go to s = Φ step. We get the minimum value for d [3, 1] (cost is 6).



Shortest Path Problem | Shortest Path Algorithms | Examples

Shortest Path Problem-

- In data structures/ADA,
- Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.
- Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.

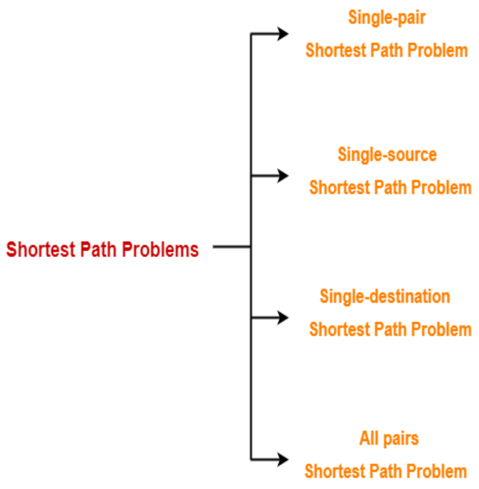
Shortest Path Algorithms-

- Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

Applications-Shortest path algorithms have a wide range of applications such as in-

- Google Maps
- Road Networks
- Logistics Research

Various types of shortest path problem are-


Shortest Path Problems
Single-pair Shortest Path Problem
Single-source Shortest Path Problem
Single-destination Shortest Path Problem
All pairs Shortest Path Problem

<u>Single-Pair Shortest Path Problem-</u>

- It is a shortest path problem where the shortest path between a given pair of vertices is computed.
- A* Search Algorithm is a famous algorithm used for solving single-pair shortest path problem.

<u>Single-Source Shortest Path Problem-</u>

- It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.
- Dijkstra's Algorithm and Bellman Ford Algorithm are the famous algorithms used for solving single-source shortest path problem.

<u>Single-Destination Shortest Path Problem-</u>

- It is a shortest path problem where the shortest path from all the vertices to a single destination vertex is computed.
- By reversing the direction of each edge in the graph, this problem reduces to single-source shortest path problem.
- Dijkstra's Algorithm is a famous algorithm adapted for solving single-destination shortest path problem.

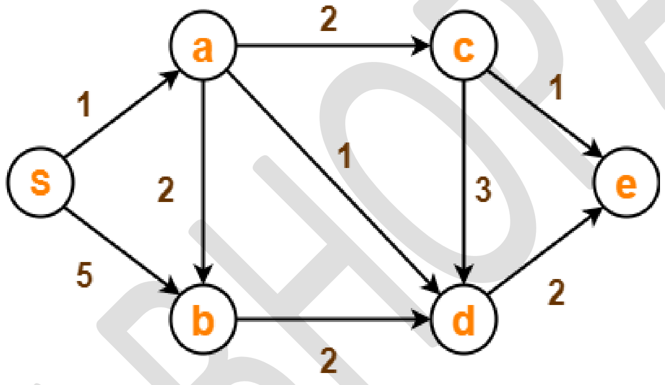
All Pairs Shortest Path Problem-

- It is a shortest path problem where the shortest path between every pair of vertices is computed.
- Floyd-Warshall Algorithm and Johnson's Algorithm are the famous algorithms used for solving All pairs shortest path problem.

Dijkstra's Algorithm

- Dijkstra Algorithm is a very famous greedy algorithm.
- It is used for solving the single source shortest path problem.
- It computes the shortest path from one particular source node to all other remaining nodes of the graph.
-

PRACTICE PROBLEM SOLUTION BASED ON DIJKSTRA ALGORITHM-



In this figure Source Vertex is "S" So Now calculate shortest path from

"S" to "a",

"S" to "b" ,

"S" to "c" ,

"S" to "d" ,

"S" to "e" ,

Initially We will calculate directed path from source vertex

Directed path from "S" to "a" is 1,

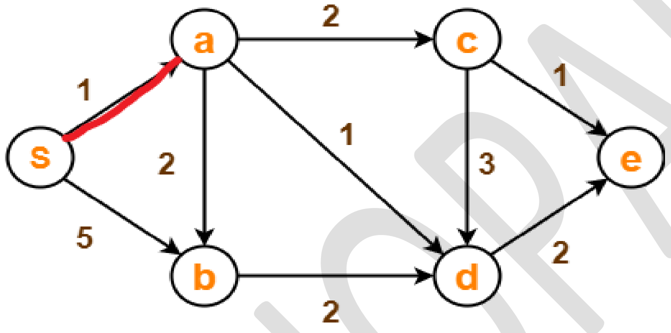
Directed path "S" to "b" is 5 ,

Directed path " S" to "c" is infinite (∞) , Directed path “S” to “d” is infinite (∞) ,

Directed path “S” to “e” is infinite (∞) ,

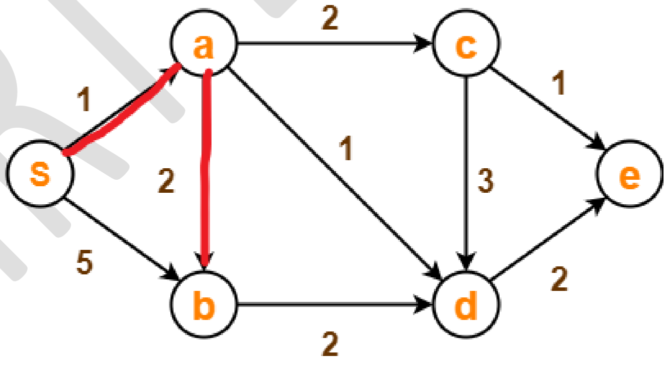
After directed path calculation from source vertex, we will try to find another path from source vertex

In-Directed path from “S” to “a” → not available. So Final Path will remain same.



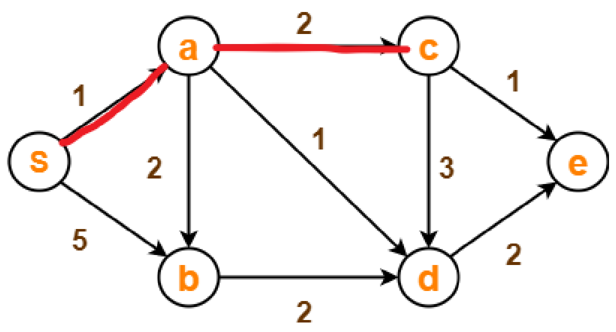
Next In-Directed path from “S” to “b”

- In-Directed path from “S” to “b” is “s to a and a to b” is 3,. Only One In-directed path is available from S to B. After that we will compare between directed path and In-directed path from “s to b” and select minimum path as a final path. So selected path from “s to b’” is 3 which is shortest path as compare directed path which is 5.



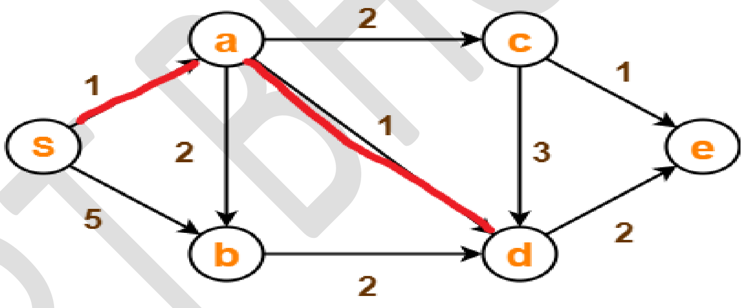
Next In-Directed path from “ S” to “c”

- Only one In-Directed path is available from “S” to “c” is “ s to a and a to c is 3. After that we will compare between directed path and In-directed path from “s to c” and select minimum path as a final path. So selected path from “s to a’” is 3 which is shortest path as compare directed path which is ∞.



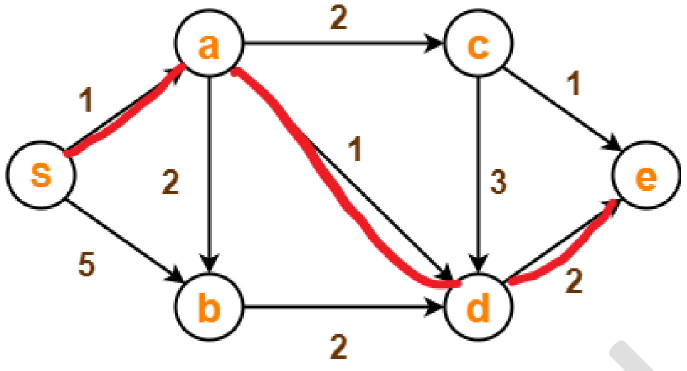
<u>Next</u> In-Directed path from " S" to "d"

- In-Directed path "S" to "d" is "s to a and a to d" is 2. Another In –Directed Path from "S" to "d" is "s to b and b to d" is 7. Another In –Directed Path from "S" to "d" is "s to a and a to b and b to d " is 5. Another In –Directed Path from "S" to "d" is "s to a and a to c and c to d " is 6
- Total 4 In-Directed path from "S" to "d" is available in the graph, so we will select shortest path which is
- "s to a and a to d" is 2

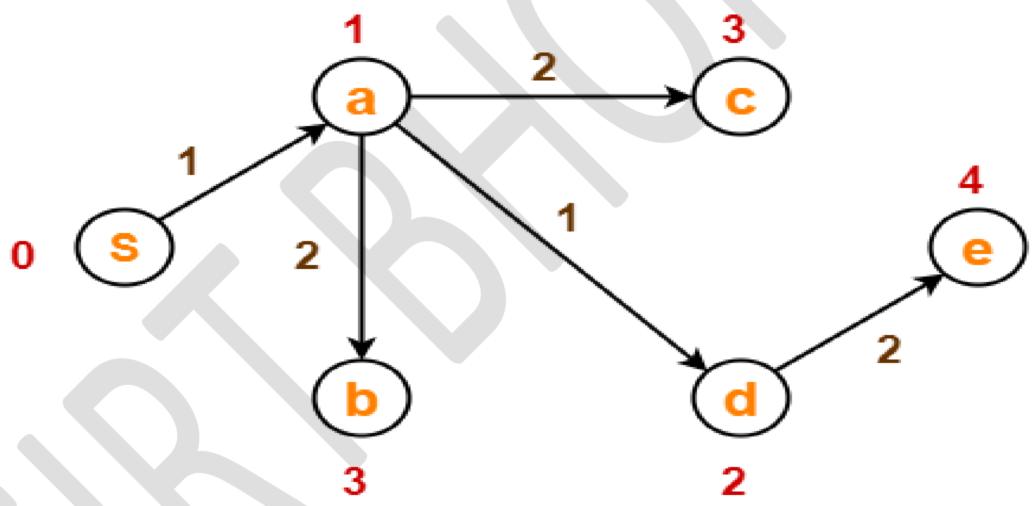


<u>Next</u> In-Directed path from " S" to "e"

- In-Directed path "S" to "e" is "s to a and a to d and d to e" is 4. Another In –Directed Path from "S" to "e" is "s to b and b to d and d to e" is 9. Another In –Directed Path from "S" to "e" is "s to a and a to b and b to d and d to e " is 7. Another In –Directed Path from "S" to "e" is "s to a and a to c and c to d and d to e " is 9. Another In –Directed Path from "S" to "e" is "s to a and a to c and c to e" is 4.
- Total 4 In-Directed path from "S" to "e" is available in the graph, so we will select shortest path which is
- In-Directed path "S" to "e" is "s to a and a to d and d to e" is 4. and Another In –Directed Path from "S" to "e" is "s to a and a to c and c to e" is 4. two path selected as minimum. Now randomly select one . Hear I selected In-Directed path "S" to "e" is "s to a and a to d and d to e" is 4.



- Now,
- All vertices of the graph are processed.
- Our final shortest path tree is as shown below.
- It represents the shortest path from source vertex 'S' to all other remaining vertices.



Shortest Path Tree

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 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 require a set of constraints divided into two categories: explicit and implicit constraints.

Explicit constraints:

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

Implicit constraints:

- Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.
- Thus, implicit constraints describe the way in which the xi's must relate to each other

Example:

For 4-queens problem: Let us consider, N = 4. Then 4-Queens Problem is to place eight queens on an 4 x 4 chessboard so that no two "attack", that is, no two of them are on the same row, column, or diagonal see Figure 1. All solutions to the 4-queens problem can be represented as 4-tuples (x1, ... , x4), where xi is the column of the ith row where the ith queen is placed.

In this problem **Explicit** and **Implicit constraints** are as:

- **Explicit constraints** using 4-tuple formation, for this problem are S= {1, 2, 3, 4}.
- **Implicit constraints** for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal

The following figure1 illustrates a solution to the 4-Queens Problem: none of the 4 queens can capture each other.

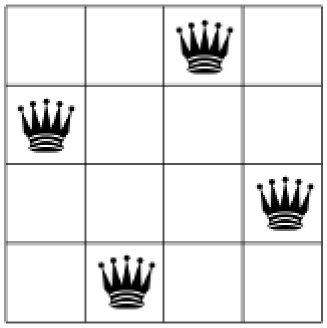


Figure 1: Soution of 4 X 4 Queen problem

Backtracking is a form of recursion. The usual scenario is that we are faced with a number of options, and we must choose one of these. After we make choice we will get a new set of options; just what set of options we get depends on what choice we made. This procedure is repeated over and over until we reach a final state or final result. If we made a good sequence of choices, we achieve final state is a *goal state;* if we didn't, it isn't.

To better understand the concept of backtracking we use an example

We start at the root of a tree (see figure 2); the tree probably has some good leaves and some bad leaves, though it may be that the leaves are all good or all bad. We want to get to a good leaf. At each node, beginning with the root, we choose one of its children to move to, and we keep this up until we get to a leaf. Suppose we get to a bad leaf. We can *backtrack* to continue the search for a good leaf by revoking our *most recent* choice, and trying out the next option in that set of options. If we run out of options, revoke the choice that got us here, and try another choice at that node. If we end up at the root with no options left, there are no good leaves to be found.

Example.

1. Starting at Root, our options are A and B. We choose A.
2. At A, our options are C and D. We choose C.
3. C is bad. Go back to A.
4. At A, we have already tried C, and it failed. Try D.
5. D is bad. Go back to A.
6. At A, we have no options left to try. Go back to Root.
7. At Root, we have already tried A. Try B.
8. At B, our options are E and F
9. . Try E.
10. E is good. Congratulations!

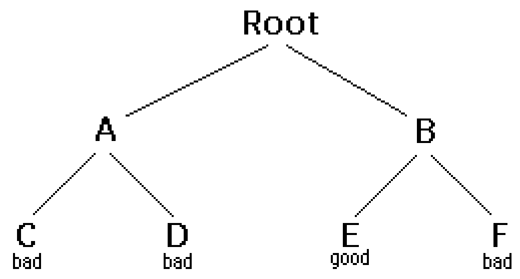


Figure 2: Tree based backtracking

In this example we drew a figure 2 of a tree. The tree is an abstract model of the possible sequences of choices we could make. There is also a data structure called a tree, but usually we don't have a data structure to tell us what choices we have. (If we do have an actual tree data structure, backtracking on it is called *depth-first tree searching.*)

Example 1: Complete Solution for 4 Queen Problem.

Problems :

4 X 4 Queen Problem

For example we are given a 4 x 4 (4 cross 4) board and 4 queens. We have to design an algorithm to place those 4 queens in 4 x 4 cross boards, so that none of the queens are clashing (row, column and diagonal wise). The will be the minimum value of 4 X 4. to find out this we start through study on queen problem are as follow

- If the board is 1 x 1, then I can place the 1 queen at (0, 0) grid.
- If board is 2 x 2, then I cannot place 2 queens. They will definitely clash, and no solution exists for this.
- For board 3 x 3, also it's not possible to get the gird arrangement of queens.
- For 4 x 4, I can place four queens at (0, 1) , (1, 3), (2, 0), (3, 2).
- Check the size of the grid as 4, return the possible arrangement.

Algorithm for (4 X 4) queen : Back-tracking algorithm for 4 X 4 queen problem is as follow:

1. Start in the topmost row.
2. If all queens are placed return true
3. Try all columns in the current row.

Do the following for every tried column:

a). If the queen can be placed safely in this column then mark this[row, column] as part of the solution and recursively check if placing queen here leads to a solution.

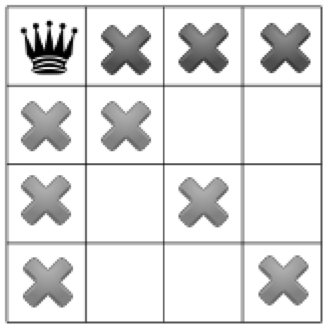
b). If placing queen in [row, column] leads to a solution then return true.

c). If placing queen doesn't lead to a solution then unlock this [row, column] (backtrack) and go to step (a) to try for other columns.

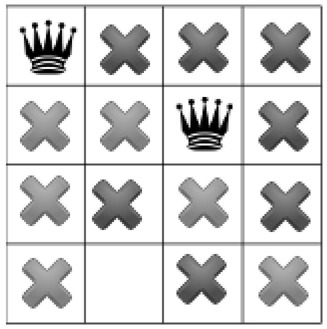
4. If all the columns have been tried and nothing worked, return false to trigger backtracking.

Solution Approach:

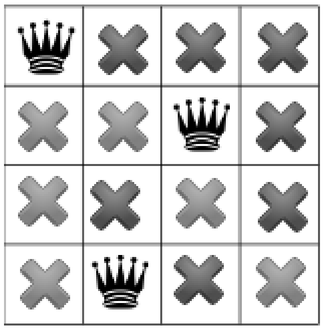
Step 1 : After placing 1st Queen at first row and first column

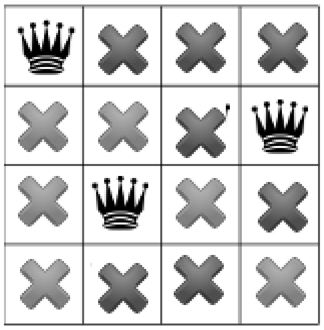


Step 2: After placing 2nd Queen at second row and third column



Step 3: It is clear from step 2 that we are not avail to place all 4 queens on 4 X 4 cross board. Only 3 queens can be placed at a time.





Step 4: Now, we have a failure at step 3 as there is no possibility to place a 4th queen in the third row (figure a) and similarly it is not possible to place 4th queen in the fourth row: there simply cannot be a solution with this configuration. The solver has to backtrack till step 1 and change the position of 1st queen.

Step 5: to find out perfect results the solver should continue the search, it would have to backtrack and try to place the queen in the right row and column. After repeating the same process so many time to get final results which is