Unit 5

CI43/CY43

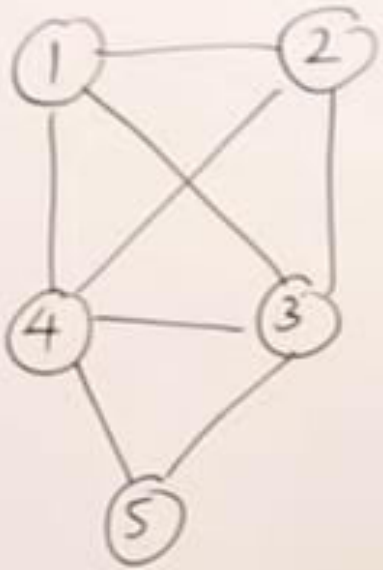
Design and Analysis of Algorithms

Hamiltonian cycle/circuit

- *The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node.*
- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- We can define the constraint for the Hamiltonian cycle problem as follows:
  - In any path, vertex i and (i + 1) must be adjacent.
  - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
  - Vertex i must not appear in the first (i – 1) vertices of any path.
- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

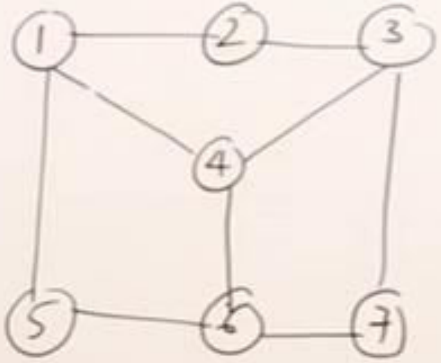
Hamiltonian Circuit(HC) examples

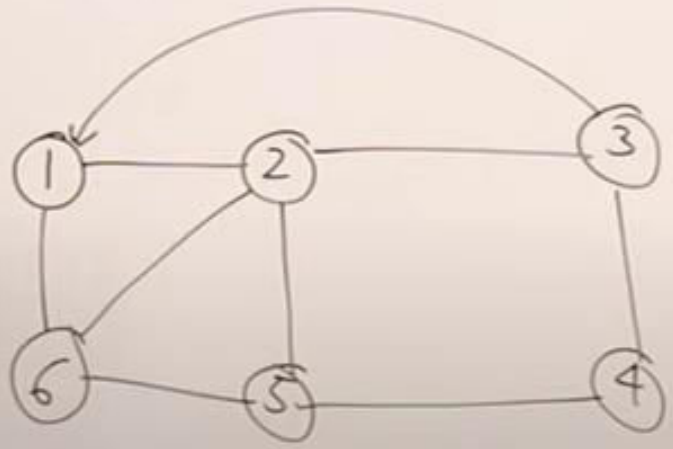
Start vertex wont be given. Set a start vertex and proceed



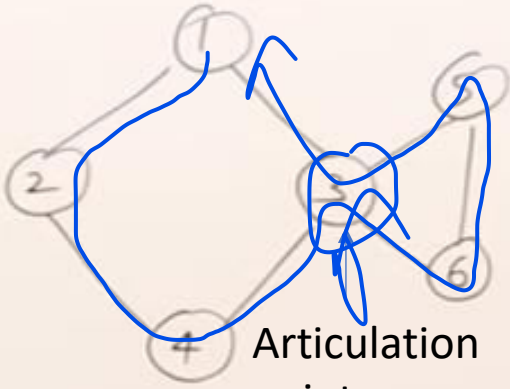
1. Start vertex 1. it can have HC like 1,2,3,5,4,1 1,2,4,5,3,1 etc

2. No HC, because node 4 cant be reached or if node 4 is reached node 7 cant be reached



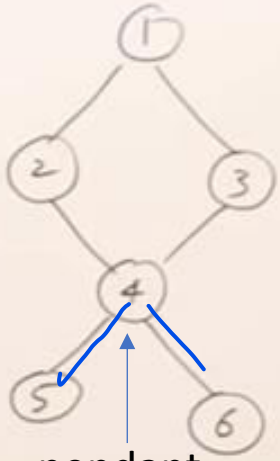


3. Start vertex 1. HC are 1,2,3,4,5,6,1 / 1,6,5,4,3,2,1



Articulation point

4. No HC possible, since articulation point exists



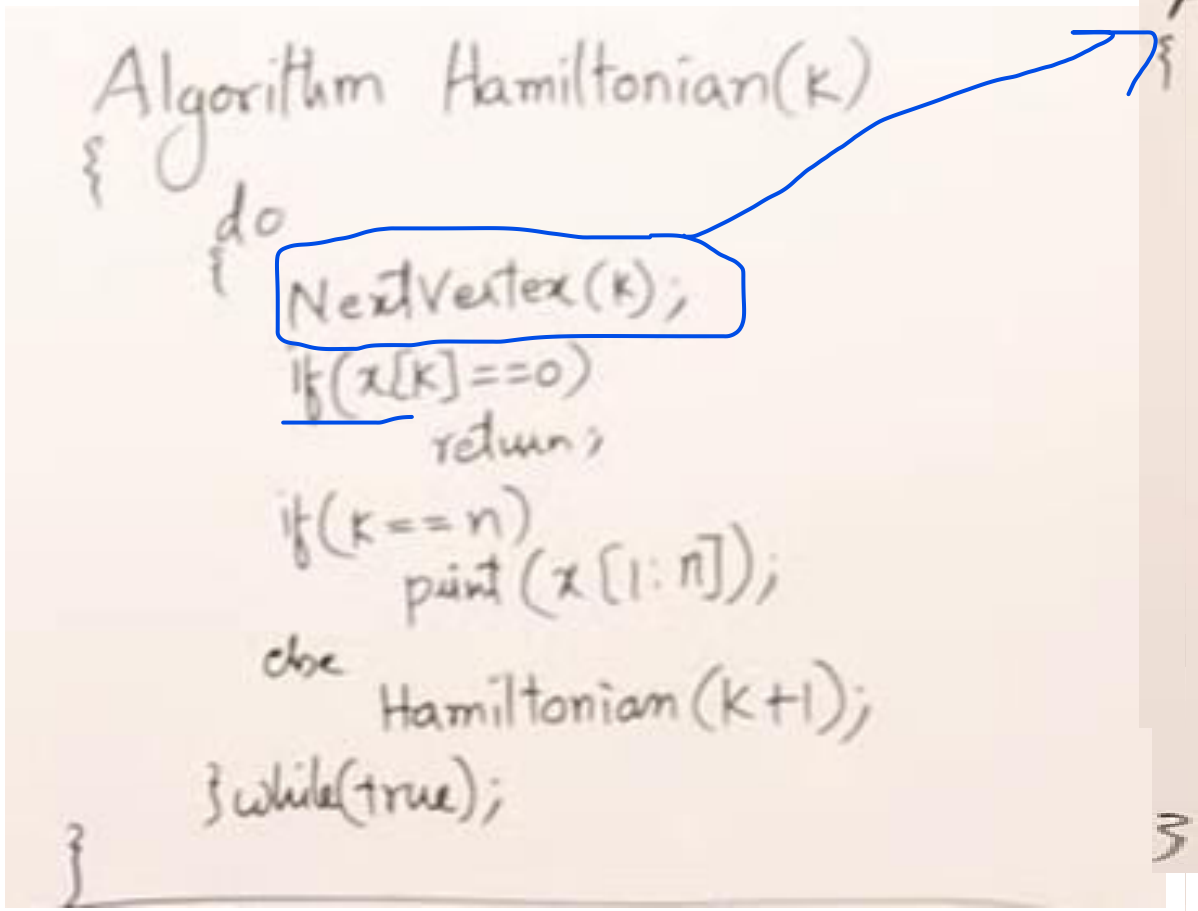
pendant

5. No HC possible since pendant Graph

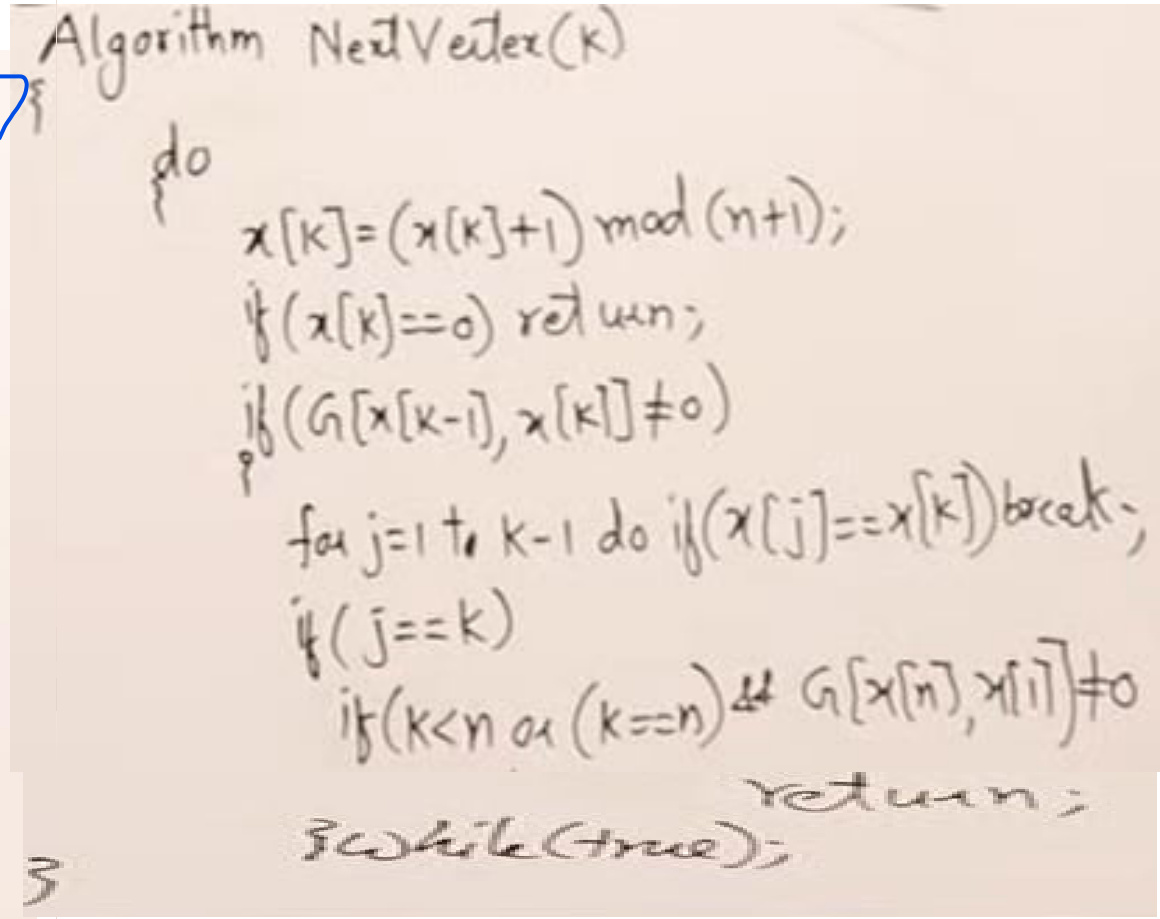
Hamiltonian cycle/circuit procedure

- Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach.
- We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed.
- The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached.
- In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed.
- The search using backtracking is successful if a Hamiltonian Cycle is obtained.

Algorithm



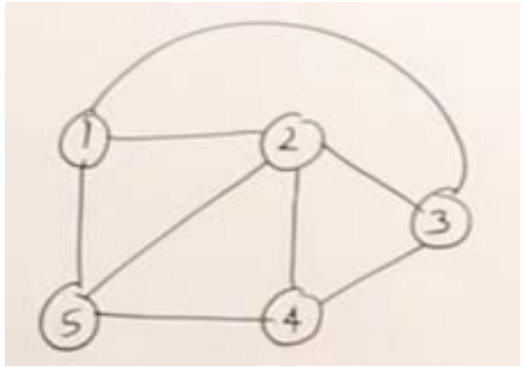
```
Algorithm Hamiltonian(k)
{
    do
    {
        NextVertex(k);
        if(x[k]==0)
            return;
        if(k==n)
            print(x[1:n]);
        else
            Hamiltonian(k+1);
    }while(true);
}
```



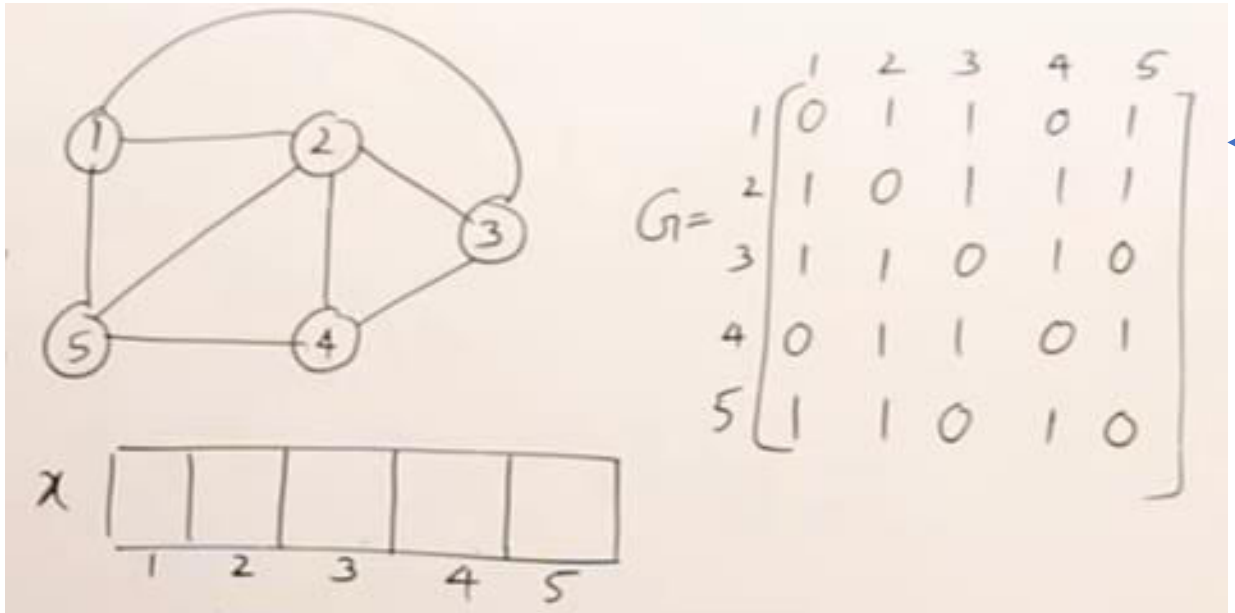
```
Algorithm NextVertex(k)
{
    do
    {
        x[k]=(x[k]+1) mod (n+1);
        if(x[k]==0) return;
        if(G[x[k-1],x[k]]≠0)
        {
            for j=1 to k-1 do if(x[j]==x[k])break;
            if(j==k)
                if(k<n or (k==n) && G[x[n],x[1]]≠0
                    return;
        }while(true);
}
```

Working of HC algorithm

- No start vertex will be given in question. Assume start vertex as 1 and proceed.
- Initial graph



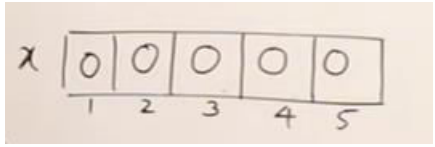
- Step 1: Set an Array to check whether all nodes are visited atleast once.



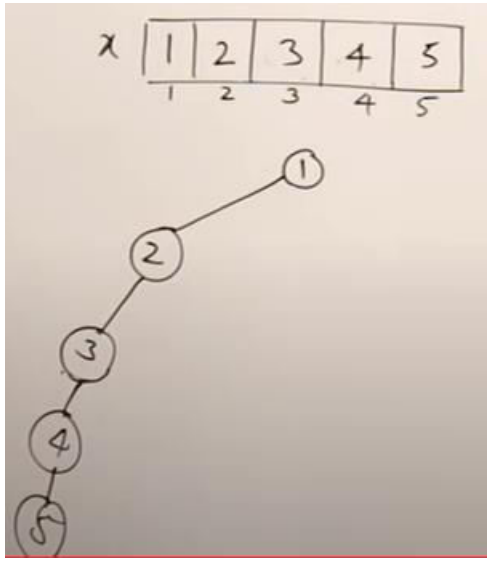
Adjacency Matrix

Steps of working of HC algorithm

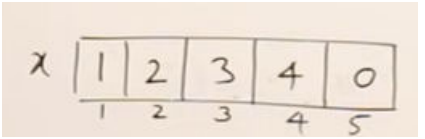
- Step 2: Initializing the array

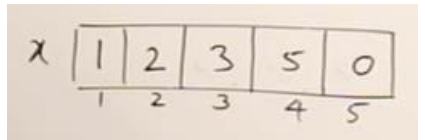


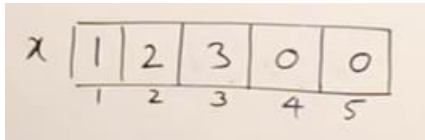
- Step 3: Creating State Space tree

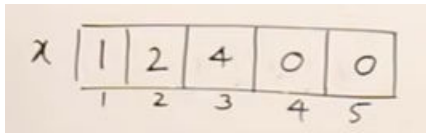


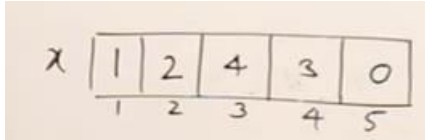
Step 4: Checking for the possibilities of HC









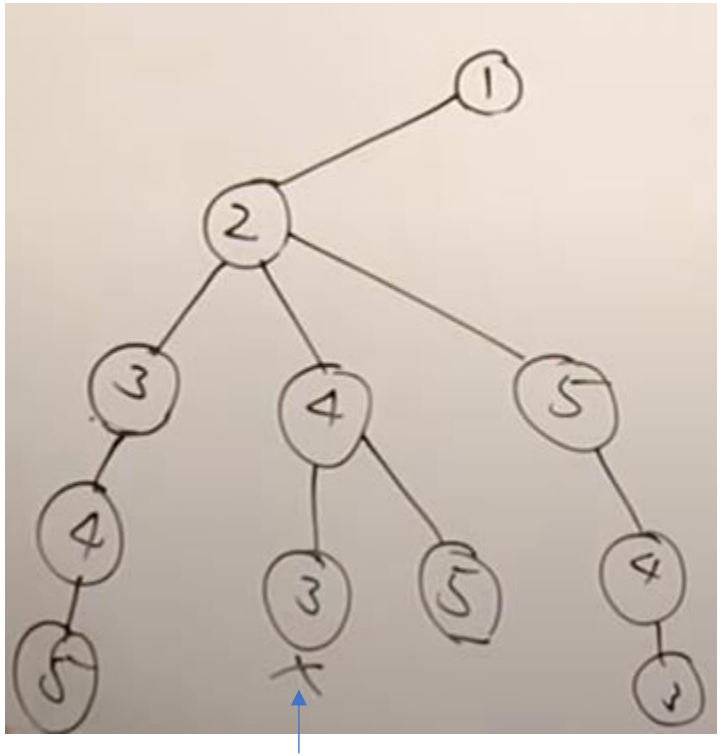


x

1	2	4	3	5

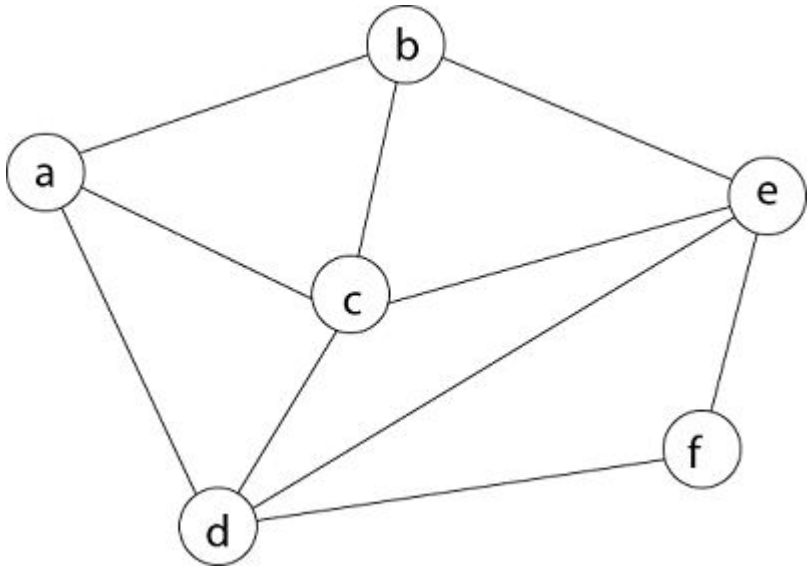
State Space Tree for Hamiltonian Circuit

- Here the HC's are 1,2,3,4,5,1 and 1,2,5,4,3,1



Try This

- Question: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



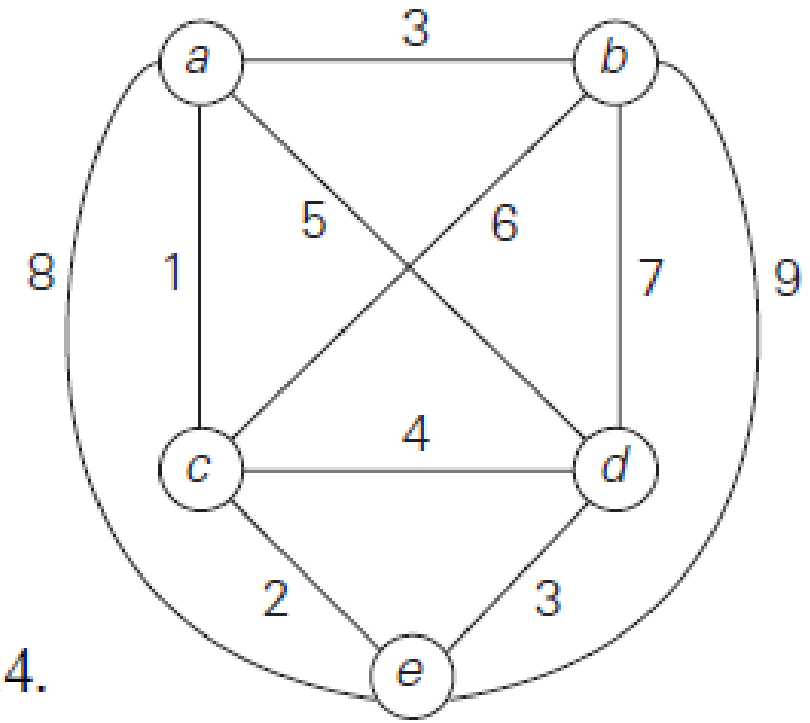
- Solution: Assume Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.

Travelling Salesman Problem(TSP) using Brand and Bound Technique

- For each city *i*, 1≤ *i* ≤ *n*, find the sum *si* of the distances from city *i* to the two nearest cities; compute the sum *s* of these *n* numbers, divide the result by 2, and, if all the distances are integers round up the result to the nearest integer:
- Lower Bound calculation: $lb = \lceil s/2 \rceil.$
- Start vertex wont be given, Assume any Start vertex and continue. Here start vertex ‘a’ is considered.

From each vertex minimum 2 cost should be considered for computing lower bound lb.

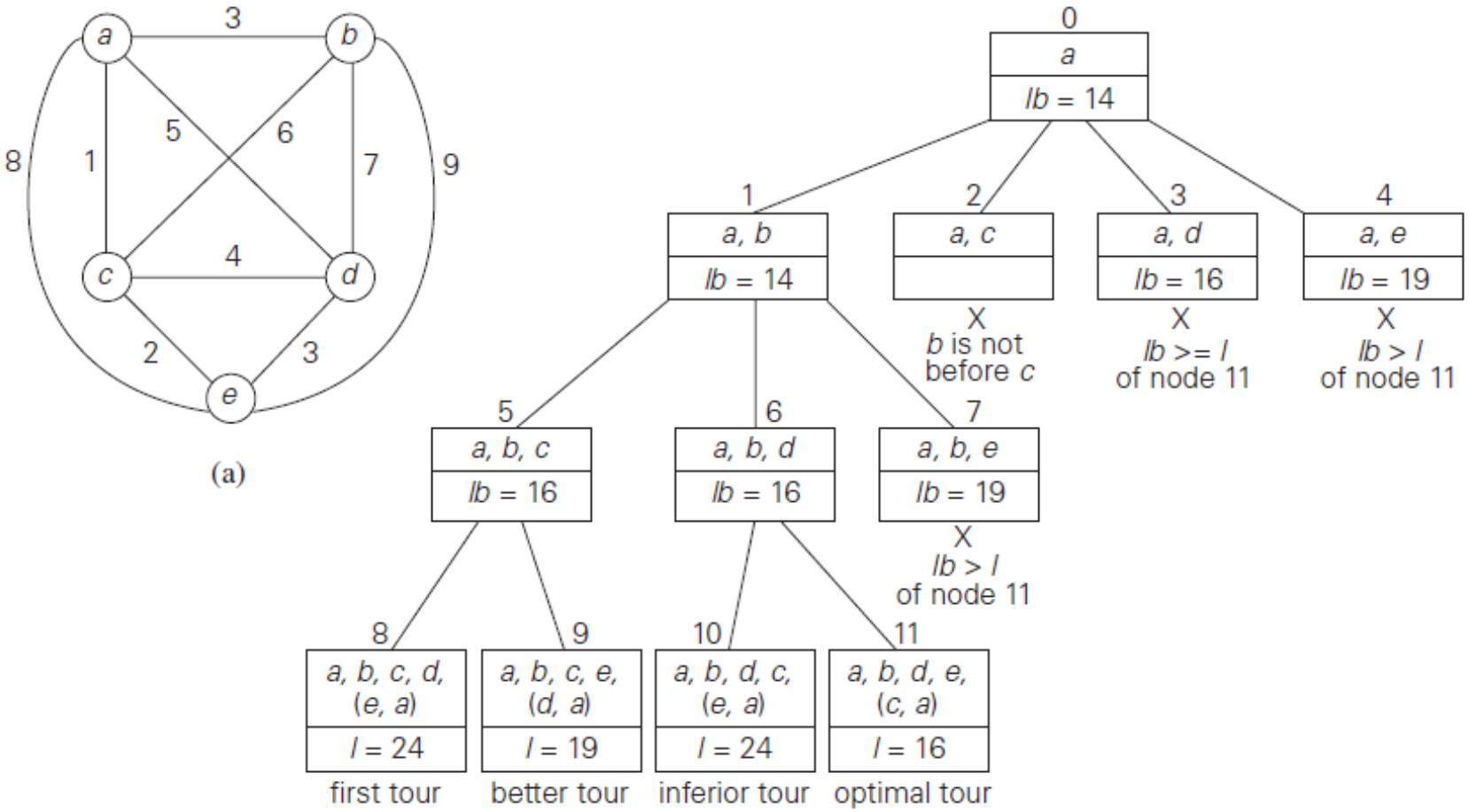
$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$



- State space tree should be constructed for solving TSP.
- Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound accordingly. For example, for all the Hamiltonian circuits of the graph that must include edge (a, d), we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges (a, d) and (d, a):
- Given below is the lower bound computation for the path (a,d)

$$\lceil [(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2 \rceil = 16.$$

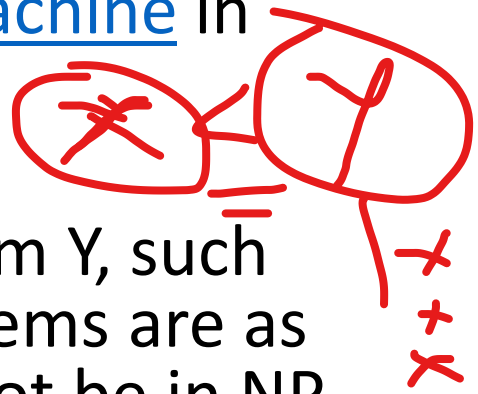
State Space Tree for TSP



Note:
Start vertex wont be specified.
Assume any start vertex and proceed.
Here the assumed start vertex is ‘a’.

NP Problem

- **NP Problem:**
  The NP problems set of problems whose solutions are hard to find but easy to verify and are solved by Non-Deterministic Machine in polynomial time.
- **NP-Hard Problem:**
  A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.
- If every problem of NP can be polynomial time reduced to it called as NP Hard.



P Class

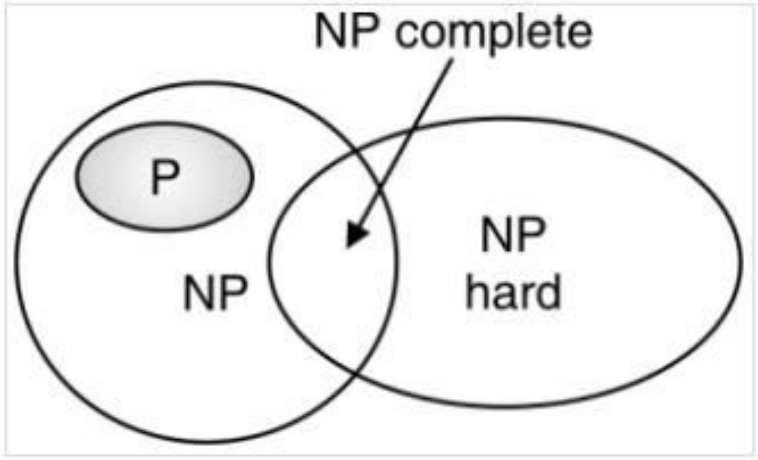
- **P Class**
- The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.
- **Features:**
- The solution to P problems is easy to find.
- P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.
- This class contains many natural problems:
- **Calculating the greatest common divisor.**
- **Finding a maximum matching.**
- **Decision versions of linear programming.**

NP Class

- The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- Features:
- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.
- Example:
- Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to personal reasons.
- This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.
- This class contains many problems that one would like to be able to solve effectively:
- Boolean Satisfiability Problem (SAT).
- Hamiltonian Path Problem.
- Graph coloring.

NP Complete

- A problem is NP-complete if the problem is both – NP-hard, and – NP.
- A language B is NP-complete if it satisfies two conditions
- B is in NP
- Every A in NP is polynomial time reducible to B.



NP-hard and NP-complete

- A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete.** The phenomenon of NP-completeness is important for both theoretical and practical reasons.

NP-complete problems

- Some of the well-known NP-complete problems are listed here :Boolean satisfiability problem.
- Knapsack problem.
- Hamiltonian path problem.
- Travelling salesman problem.
- Subset sum problem.
- Vertex covers the problem.
- Graph colouring problem.

Comparison between NP-hard and NP-Complete

Parameters	NP-Hard Problem	NP-Complete Problem
Meaning and Definition	One can only solve an NP-Hard Problem X only if an NP-Complete Problem Y exists. It then becomes reducible to problem X in a polynomial time.	Any given problem X acts as NP-Complete when there exists an NP problem Y- so that the problem Y gets reducible to the problem X in a polynomial line.
Presence in NP	The NP-Hard Problem does not have to exist in the NP for anyone to solve it.	For solving an NP-Complete Problem, the given problem must exist in both NP-Hard and NP Problems.
Decision Problem	This type of problem need not be a Decision problem.	This type of problem is always a Decision problem (exclusively).
Example	Circuit-satisfactory, Vertex cover, Halting problems, etc., are a few examples of NP-Hard Problems.	A few examples of NP-Complete Problems are the determination of the Hamiltonian cycle in a graph, the determination of the satisfaction level of a Boolean formula, etc.