SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

18CSC204J - DESIGN AND ANALYSIS OF ALGORITHMS

Unit- IV

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Graph Introduction

- A graph - an abstract data structure that is used to implement the graph concept from mathematics.
- A collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of a having a purely parent-to-child relationship between tree nodes.
- Any kind of complex relationships between the nodes can be represented.

Definition

- A graph G is defined as an ordered set (V, E), where V(G) represent the set of vertices and E(G) represents the edges that connect the vertices.
- The figure given shows a graph with V(G) = { A, B, C, D and E} and E(G) = { (A, B), (B, C), (A, D), (B, D), (D, E), (C, E) }. Note that there are 5 vertices or nodes and 6 edges in the graph.

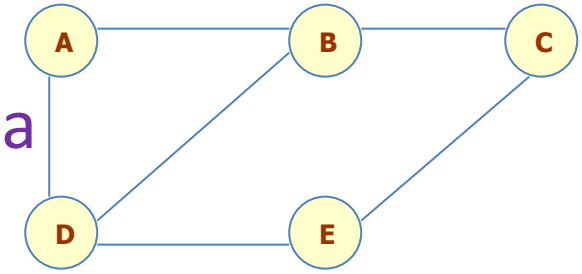


Fig:a

A graph can be directed (Fig a)or undirected (Fig b).

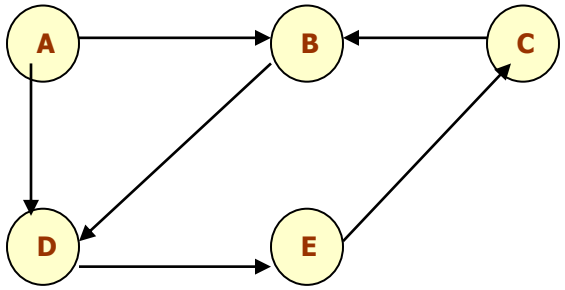


Fig:a

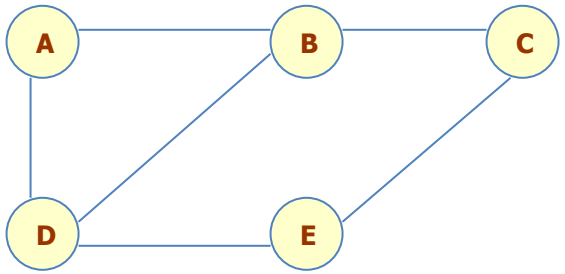


Fig:b

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

GRAPH TRAVERSAL ALGORITHMS:

GRAPH TRAVERSAL ALGORITHMS:

- This technique is used for searching a vertex in a graph.
- It is also used to decide the order of vertices to be visit in the search process.
- There are two standard methods of graph traversal:
    1. Breadth-first search
    2. Depth-first search
- While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing.
- The depth-first search scheme uses a stack.

Breadth-First Search Algorithm:

- It is a graph search algorithm that begins at the root node and explores all the neighboring nodes.
- Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Queue data structure with maximum size of total number of vertices in the graph.

Breadth-First Search Algorithm:

STEPS:

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

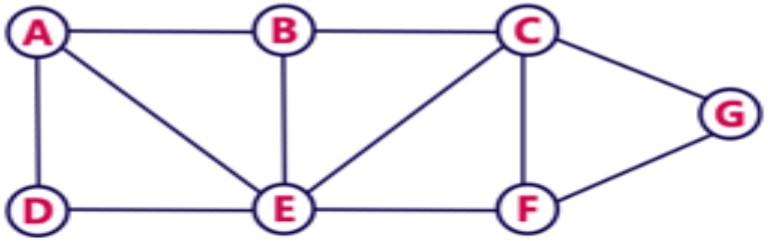
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Breadth-First Search Algorithm:

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

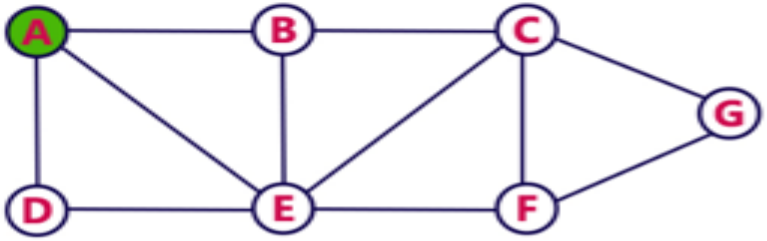
Breadth-First Search Algorithm:

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

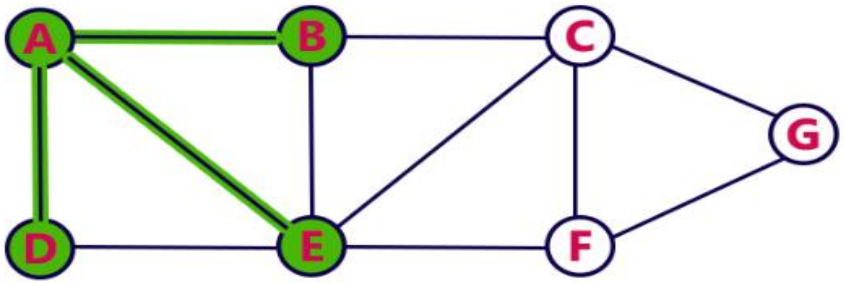


Queue

Breadth-First Search Algorithm:

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

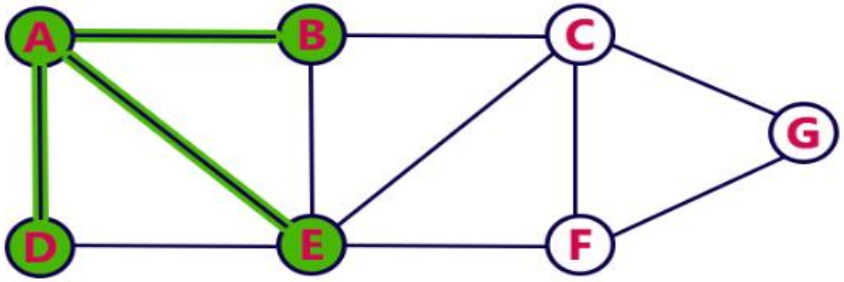


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



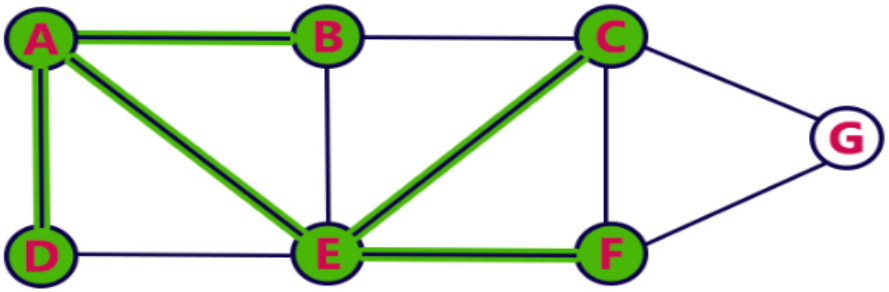
Queue

		E	B			

Breadth-First Search Algorithm:

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

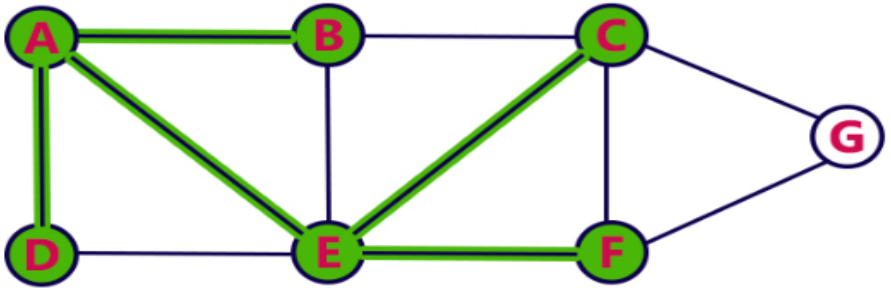


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



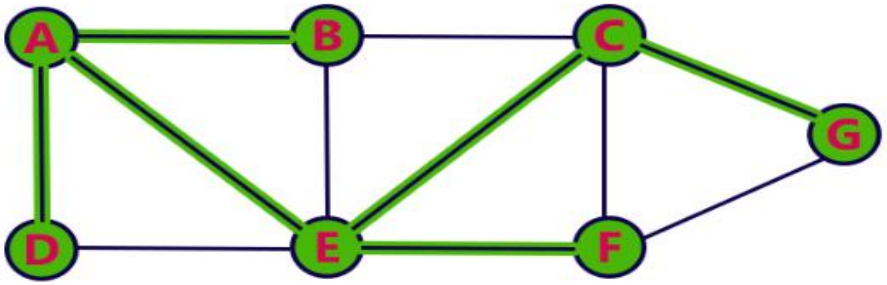
Queue

				C	F	

Breadth-First Search Algorithm:

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

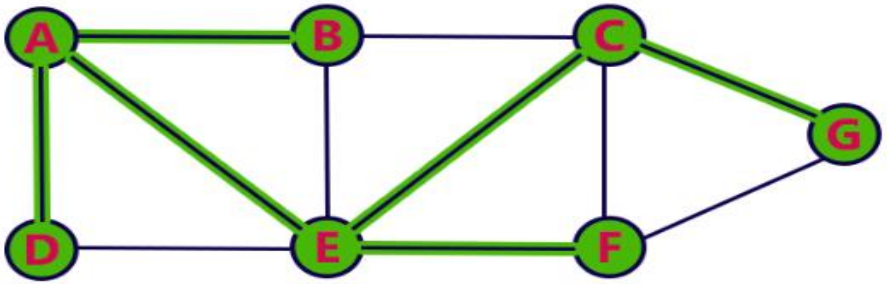


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



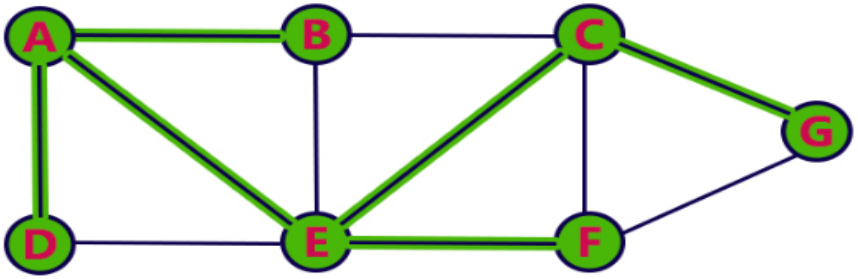
Queue

						G

Breadth-First Search Algorithm:

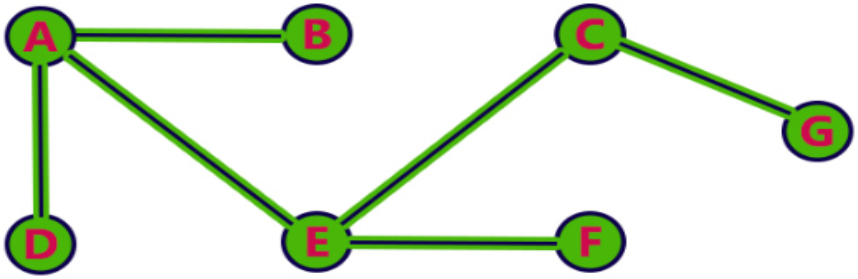
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



***Features of Breadth-First Search Algorithm*:**

- ***Space complexity***
  - In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated.
  - The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).
- ***Time complexity***
  - In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd).
  - However, the time complexity can also be expressed as O(| E | + | V |), since every vertex and every edge will be explored in the worst case.

- ***Completeness***
  - Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph.
  - But in case of an infinite graph where there is no possible solution, it will diverge.
- ***Optimality***
  - Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node.
  - But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm:

- Breadth-first search can be used to solve many problems such as:
- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth-first Search Algorithm:

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A, and so on.

Depth-first Search Algorithm:

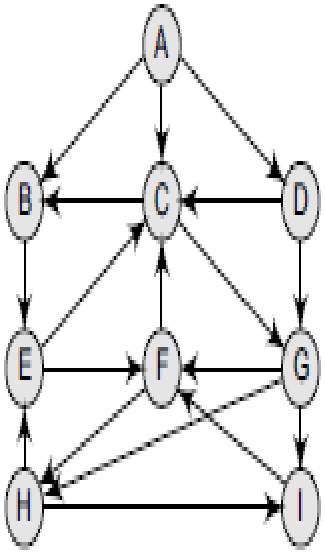
- During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- This technique is used for searching a vertex in a graph.
- It produces a spanning tree as a final result.
- Spanning tree is a graph without any loops.
- Here we use Stack data structure with maximum size of total number of vertices in the graph.

Depth-first Search Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Depth-first Search Algorithm:

Consider the graph and find the spanning tree,



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Depth-first Search Algorithm:

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

Depth-first Search Algorithm:

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G | STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B | STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E | STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Features of Depth-First Search Algorithm

Space complexity The space complexity of a depth-first search is lower than that of a breadth first search.

Time complexity The time complexity of a depth-first search is proportional to the number of
vertices plus the number of edges in the graphs that are traversed. The time complexity can be
given as (O(|V| + |E|)).

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

Introduction to Shortest Path

Shortest Path Problem-

- 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.

hortest Path Algorithms-

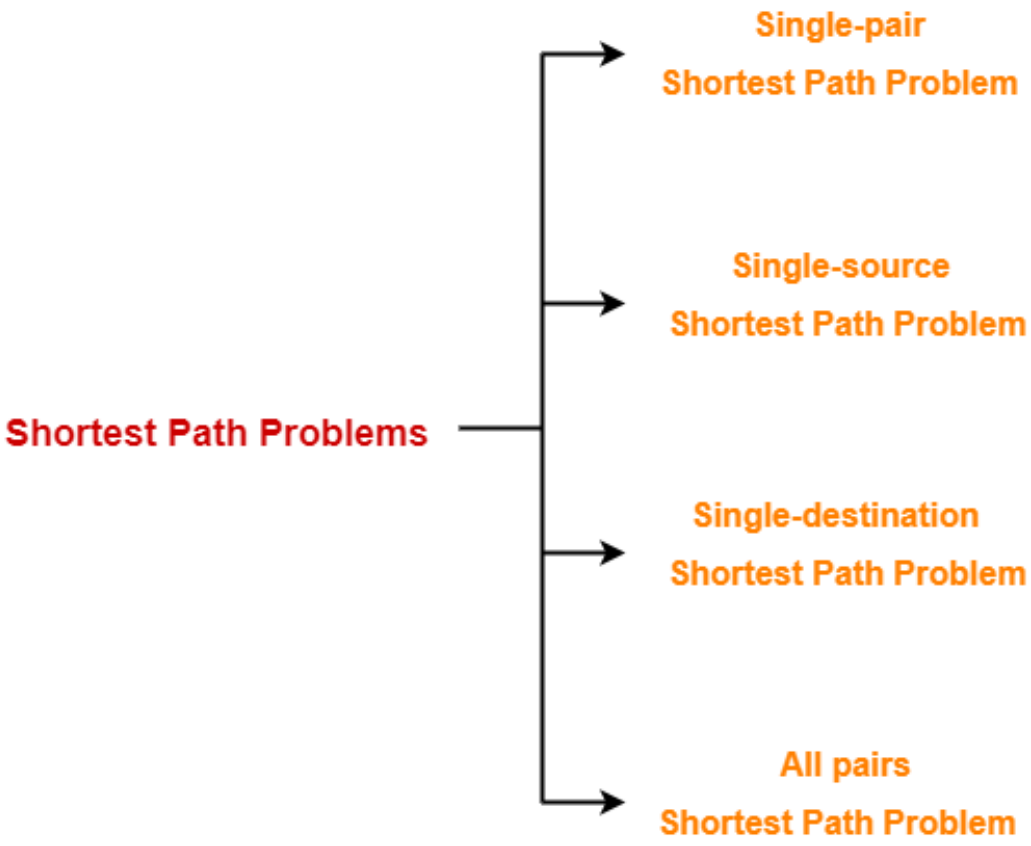
Shortest path algorithms are a family of algorithms used for solving the shortest path problem.

Applications-

Shortest path algorithms have a wide range of applications such as in-
Google Maps
Road Networks
Logistics Research

Types of Shortest Path Problem-

Various types of shortest path problem are-



Single-Pair Shortest Path Problem-

•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.

Single-Source Shortest Path Problem-

•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.

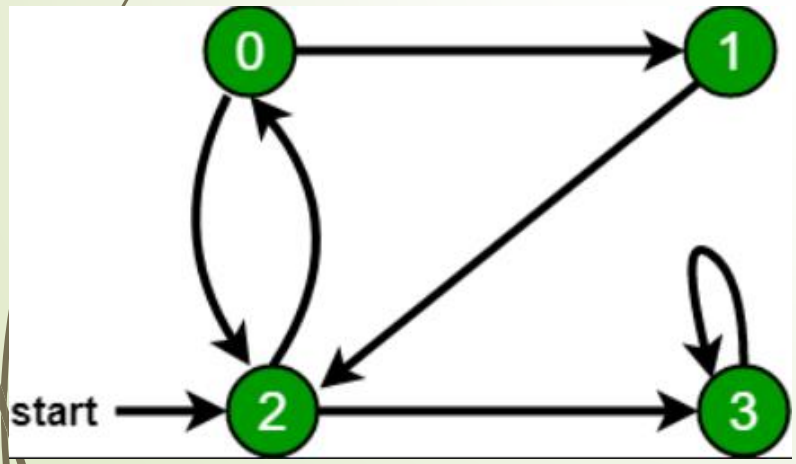
Single-Destination Shortest Path Problem-

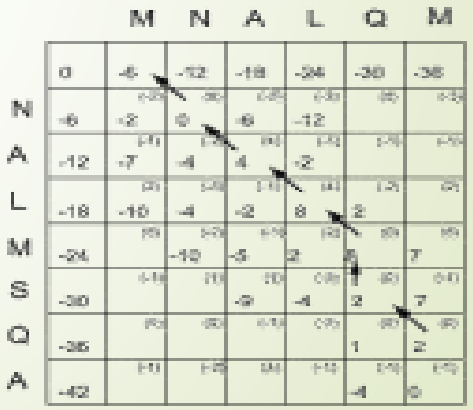
•It is a shortest path problem where the shortest path from all the vertices to a single destination vertex is computed.
•By reversing the direction of each edge in the graph, this problem reduces to single-source shortest path problem.
•Dijkstra's Algorithm is a famous algorithm adapted for solving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

Graph Introduction





Graph

- A graph is a collection of nodes and edges
- Denoted by $G = (V, E)$.

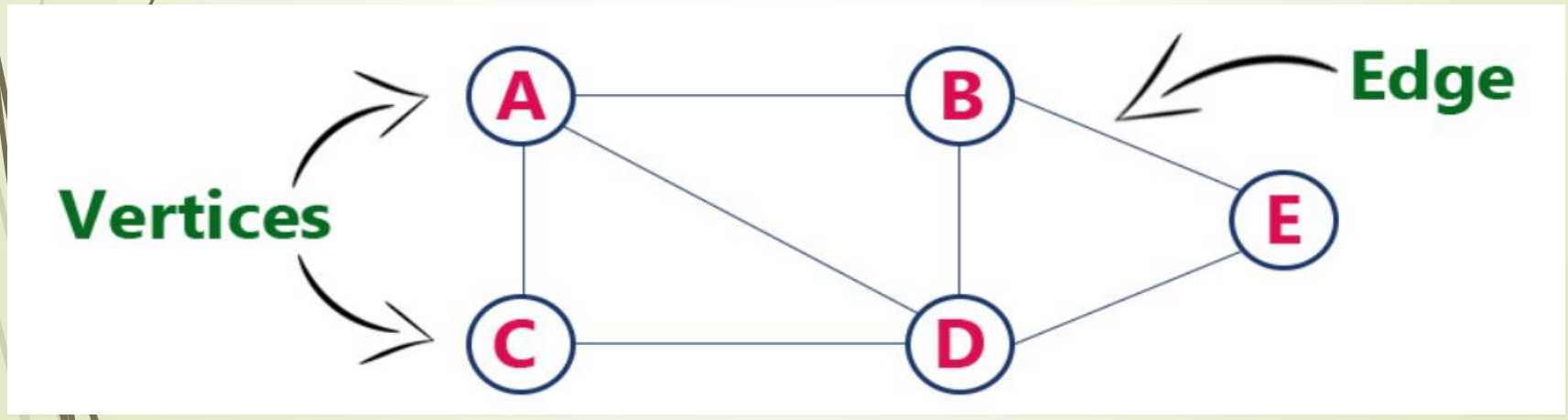
V = **nodes** (vertices, points).

E = **edges** (links, arcs) between pairs of nodes.

Graph size parameters: $n = |V|$, $m = |E|$.

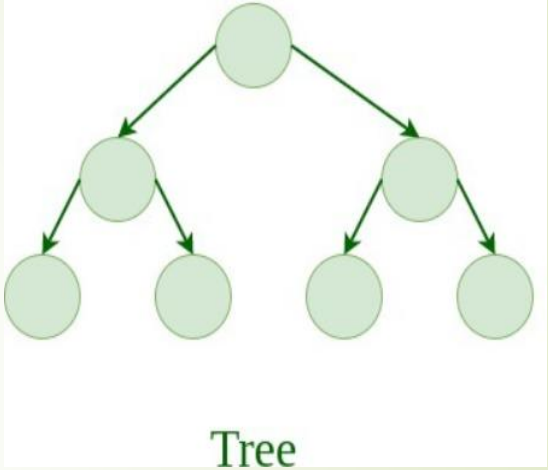
Graph Example

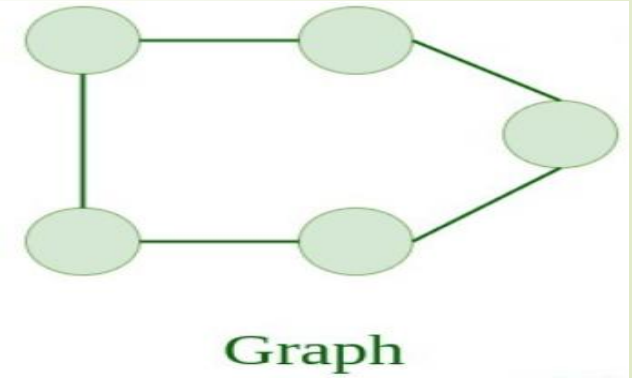
- Graph G can be represented as **G = (V , E)**
- Where **Vertices(V) = {A,B,C,D,E}**
- **Edge(E) = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.**



Tree Vs Graph

- Trees are the restricted types of graphs, just with some more rules.
- Every tree will always be a graph but not all graphs will be trees.
- Linked List, Trees, and Heaps all are special cases of graph





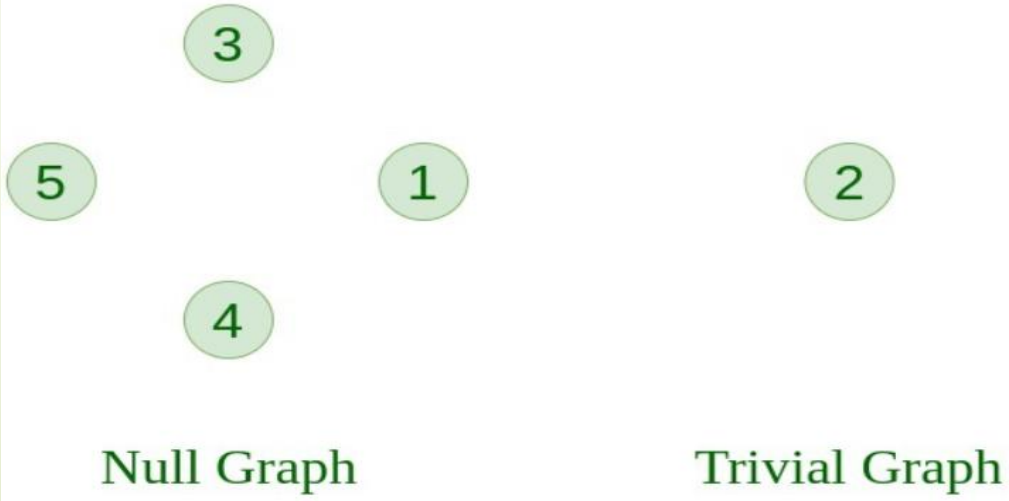
Types of Graph

Null Graph

A graph is known as null graph if there are **no edges** in the graph.

Trivial Graph

Graph having only a **single vertex**, it is the **smallest graph** possible.



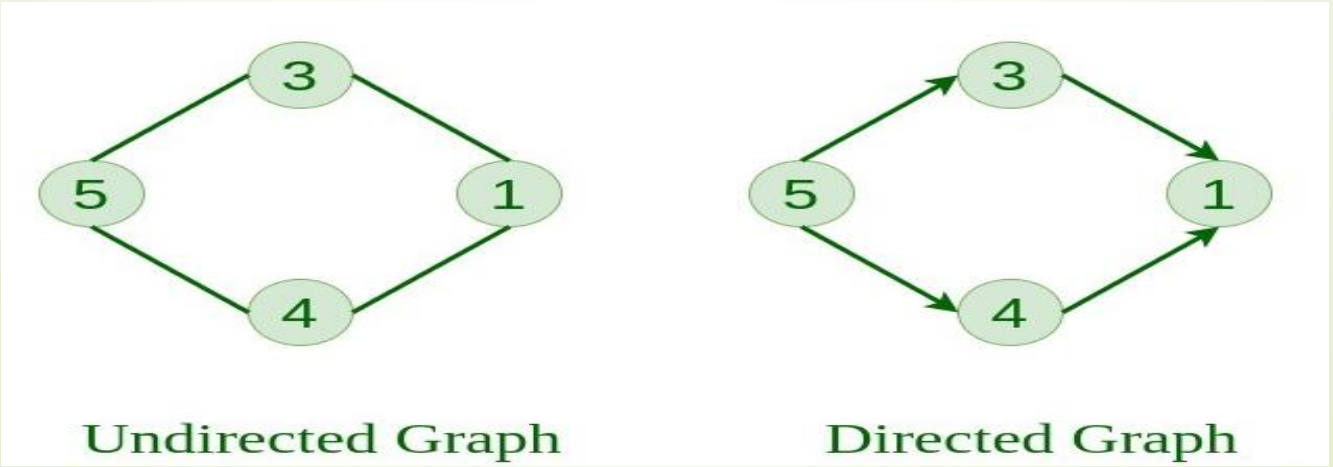
Types of Graph

Undirected Graph

A graph in which edges **do not have any direction**.

Directed Graph

A graph in which edge **has direction**.



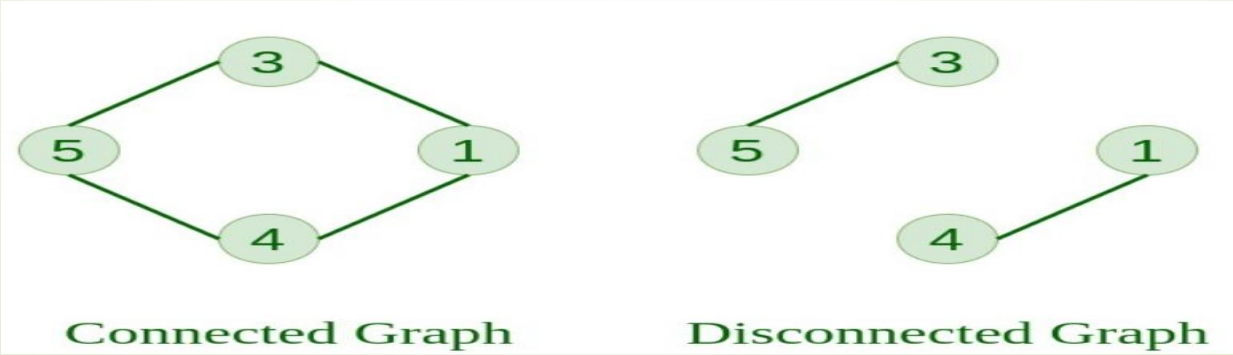
Types of Graph

Connected Graph

The graph in which from **one node we can visit any other node.**

Disconnected Graph

The graph in which **at least one node is not reachable from a node.**



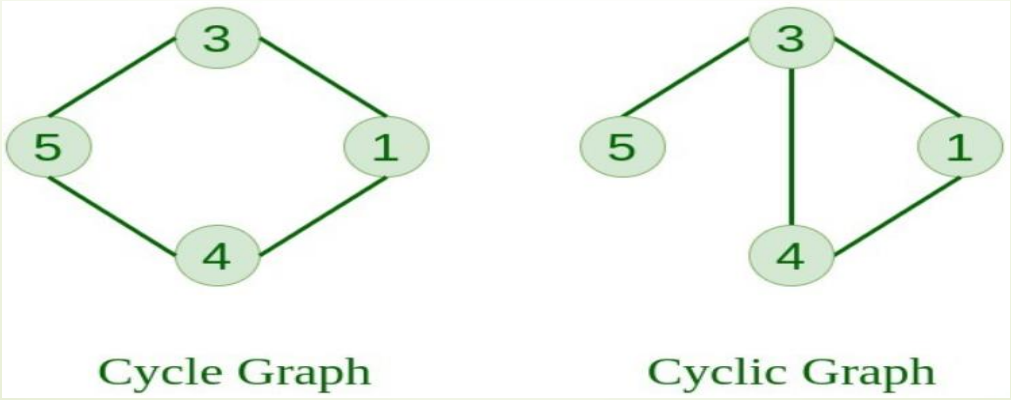
Types of Graph

Complete Graph

The graph in which **from each node there is an edge to each other node.**

Cycle Graph

The graph in which the graph is a cycle in itself, **the degree of each vertex is 2.**



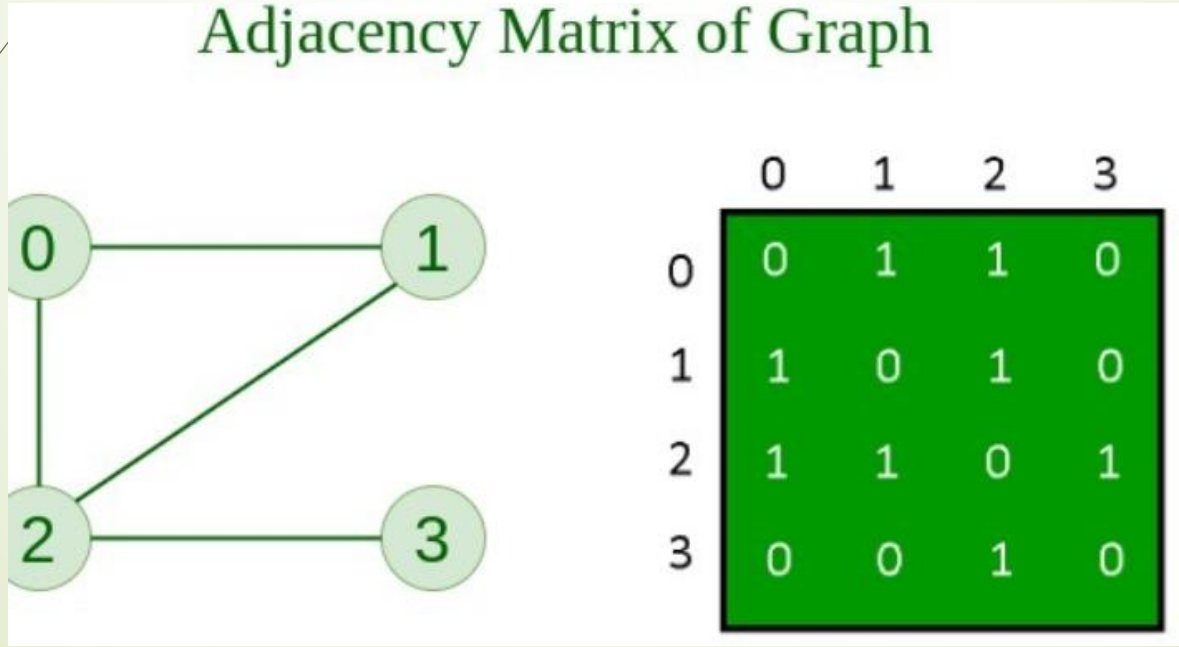
Representation of Graph

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

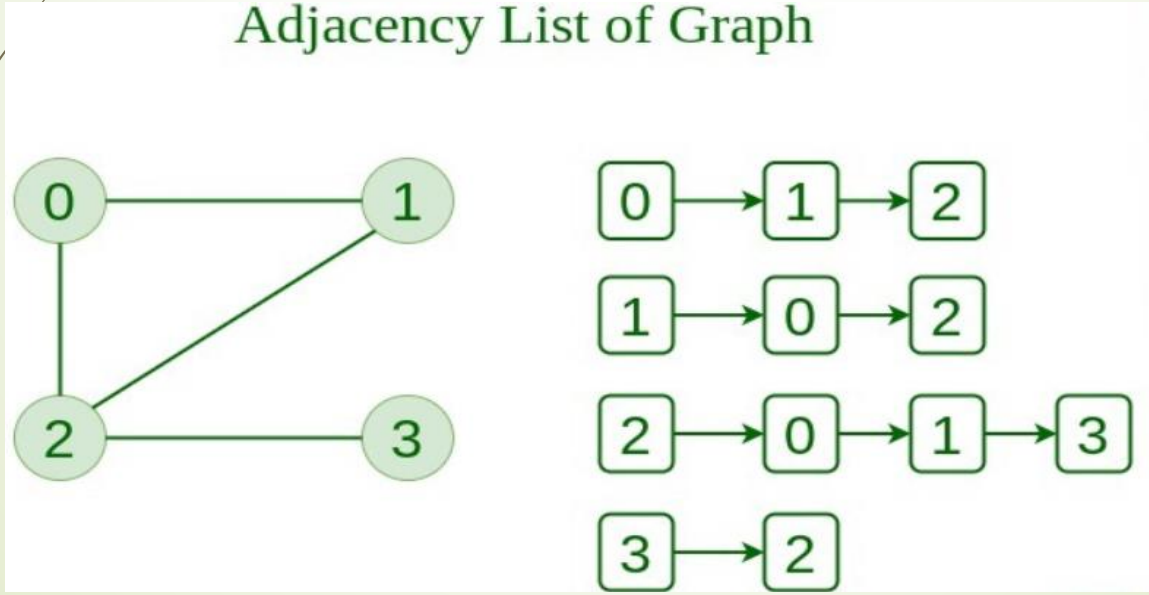
- In this method, the graph is stored in **the form of the 2D matrix** where rows and column denote vertices.



Adjacency List

- This graph is represented as a **collection of linked lists.**

  There is an array of pointer which points to the edges connected to that vertex



Breadth First Search(BFS)

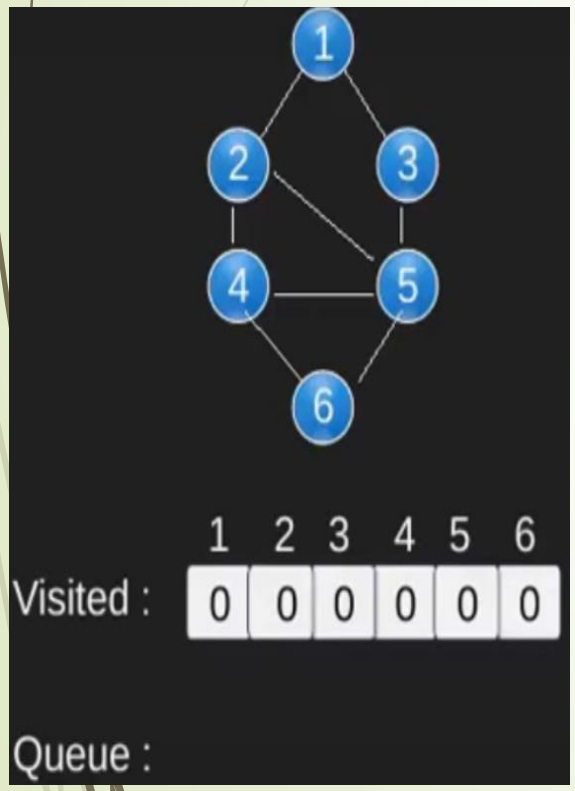
- Breadth first search is a **graph traversal algorithm** that starts traversing the graph from root node and explores all the neighbouring nodes.
- It uses a **Queue data** structure for implementation.
- Breadth-first search (BFS) is an algorithm that is used to **graph data** or **searching tree** or **traversing structures.**

Algorithm

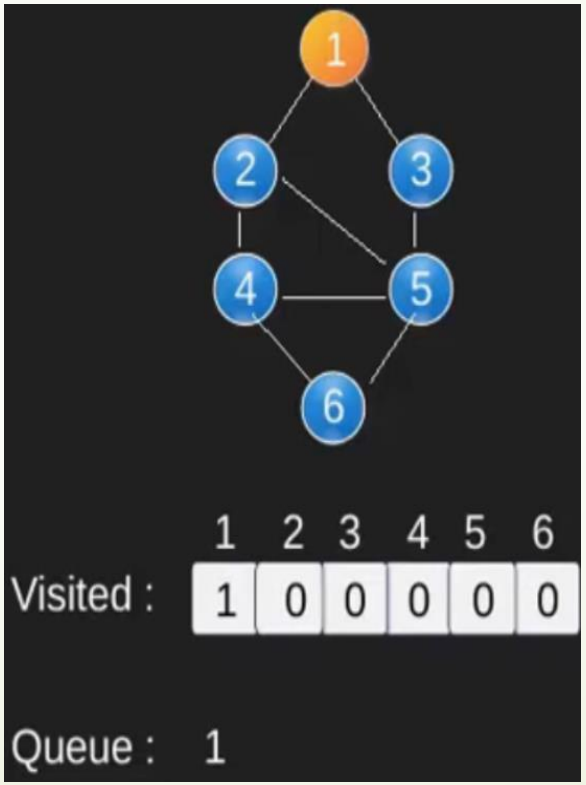
```
BFS (G, s)                    //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s )            //Inserting s in queue until all its neighbour vertices are marked.
    mark s as visited.
    while ( Q is not empty)   //Removing that vertex from queue,whose neighbour will be visited now
    v = Q.dequeue( )          //processing all the neighbours of v
    for all neighbours w of v in Graph G
    if w is not visited
            Q.enqueue( w )    //Stores w in Q to further visit its neighbour
            mark w as visited.
```

Example of BFS

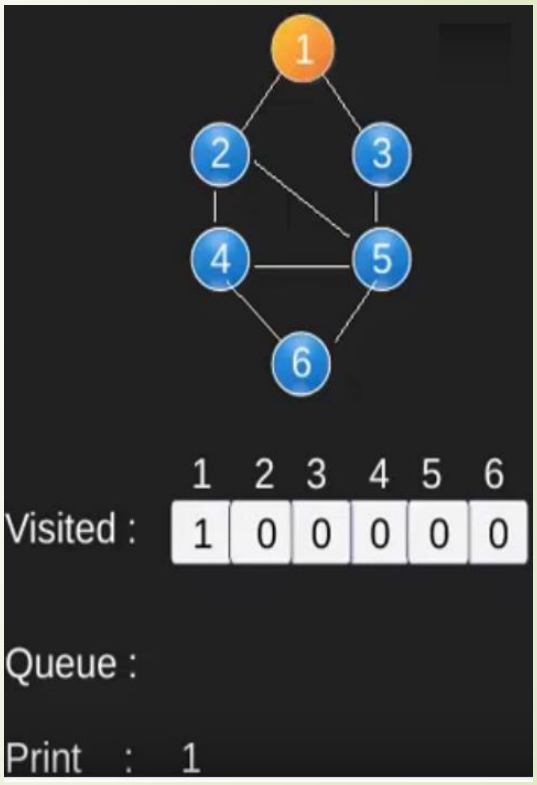
Step 1 :



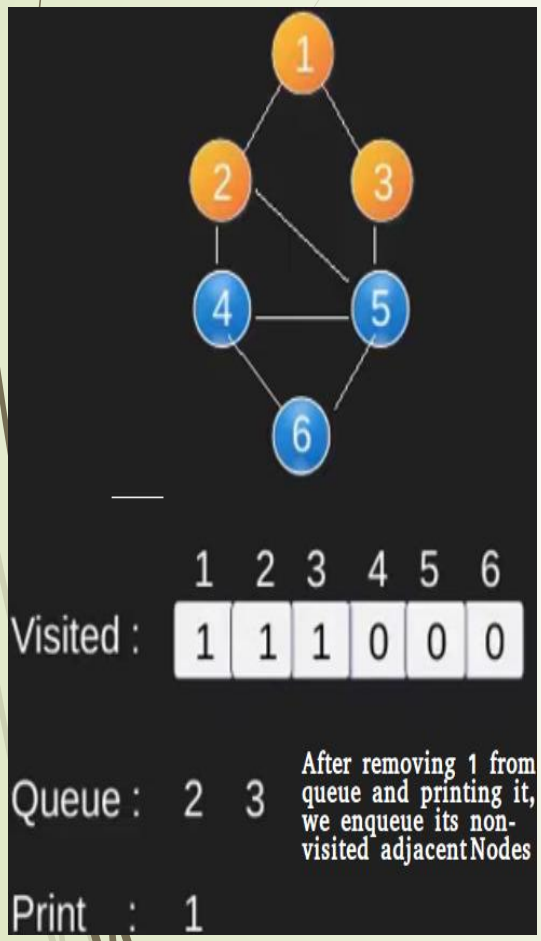
Step 2 :



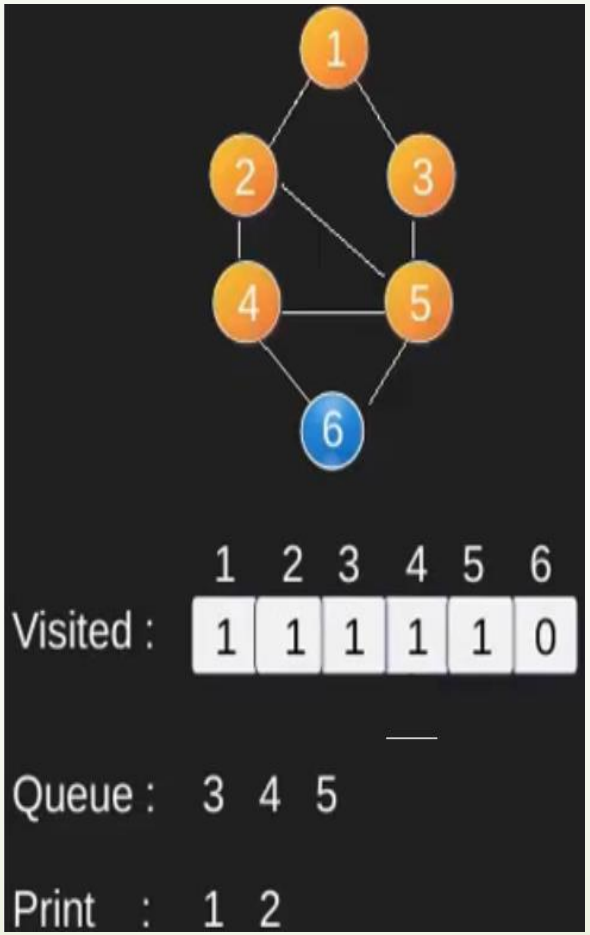
Step 3 :



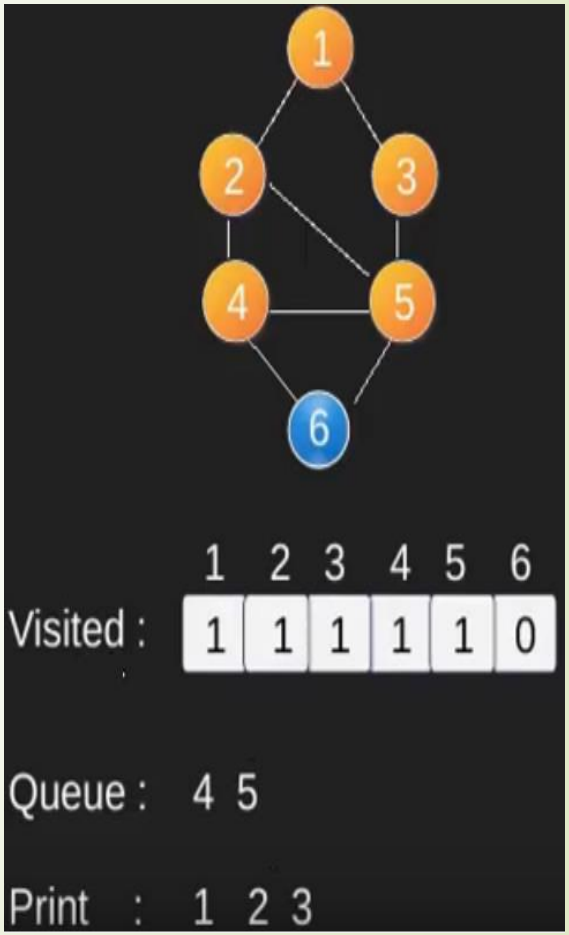
Step 4 :



Step 5 :



Step 6 :



Depth First Search(DFS)

- Depth first search (DFS) algorithm starts with the initial node of the graph G, and then **goes to deeper and deeper until we find the goal node** or **the node which has no children.**

- It is an algoirthm **for traversing the tree or graph data structure**

- Application of DFS – Topological Sorting, Finding the bridges of a graph, Cycle Detecting

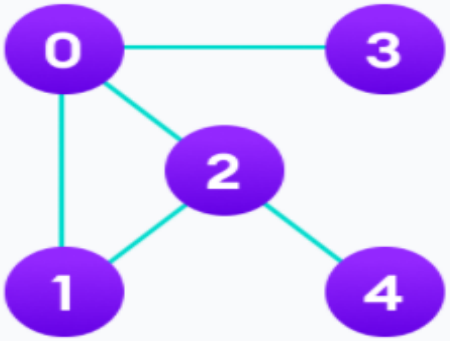
Algorithm

```
DFS-iterative (G, s):    //Where G is graph and s is source vertex
    let S be stack
    S.push( s ) //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
            //Pop a vertex from stack to visit next
            v = S.top( )
            S.pop( )
            //Push all the neighbours of v in stack that are not visited
            for all neighbours w of v in Graph G :
            if w is not visited :
```

```
        S.push( w )
    mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
        DFS-recursive(G, w)
```

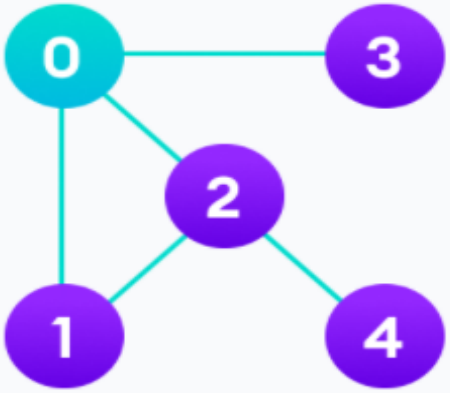
Example of DFS

Step 1 : Undirected graph with 5 vertices



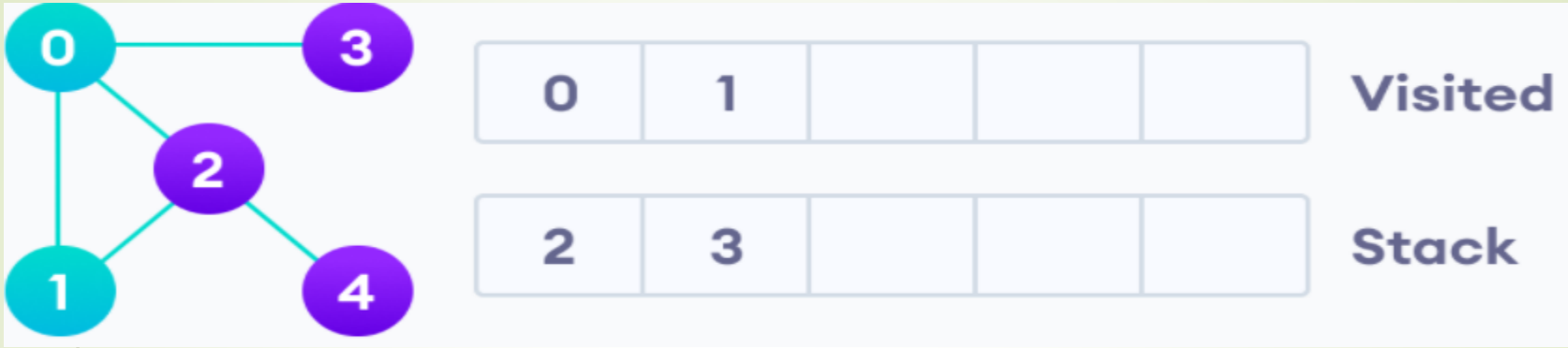
					Visited
					Stack

Step 2 : Visit the element and put it in the visited list

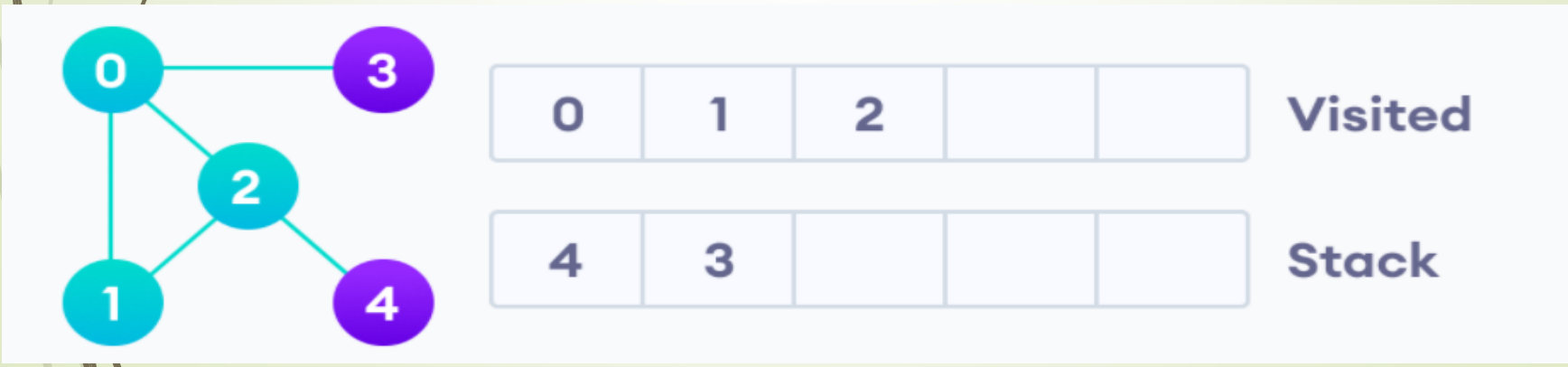


0					Visited
1	2	3			Stack

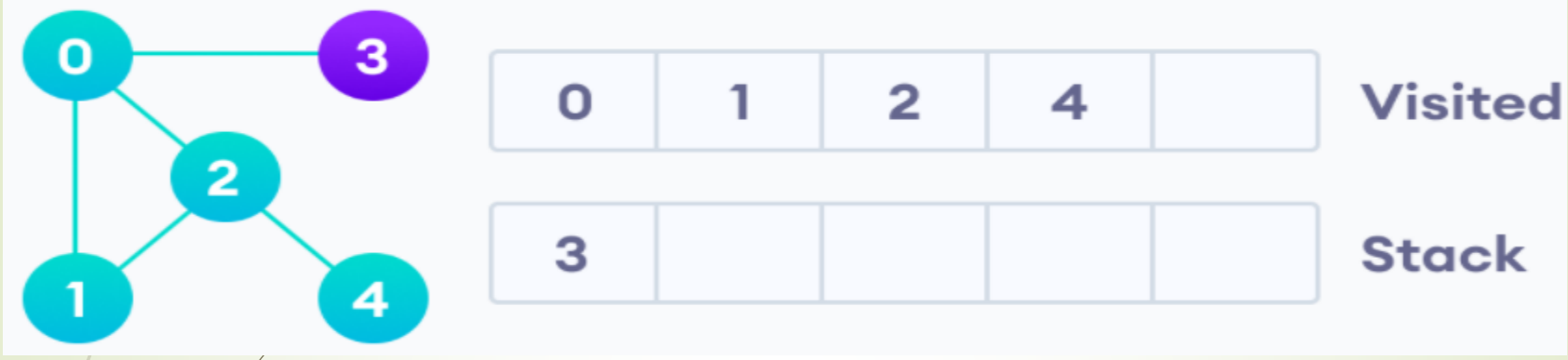
Step 3 : Visit the element at the top of stack



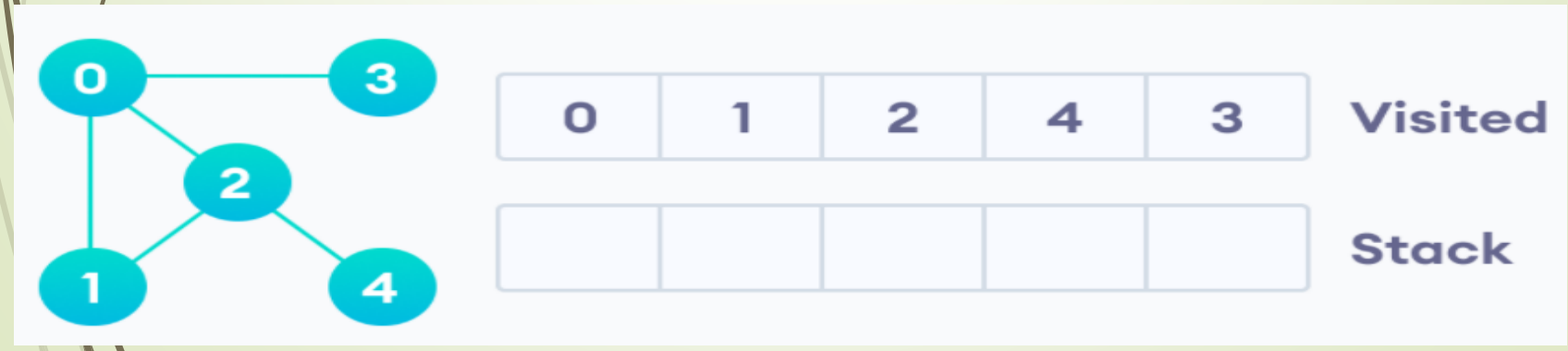
Step 4 : Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Step 5 : Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



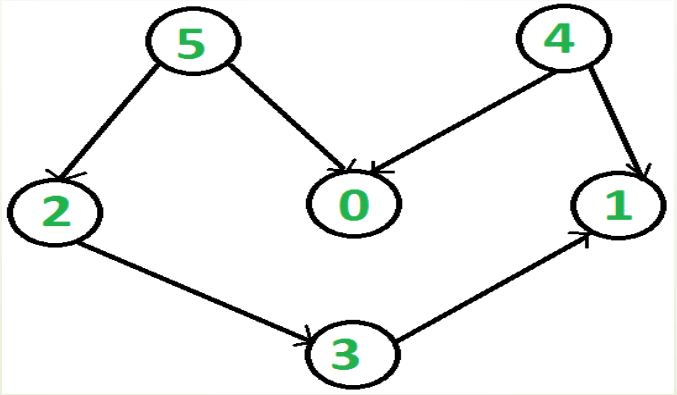
Step 6 : After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph



Topological Sorting

- Topological sorting for **Directed Acyclic Graph** (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.

- It is an application of **Breadth First Search.**



Application of Graph

- Amazon use graphs to make suggestions for future shopping
- Graph is use in solving Travelling Salesman Problem(TSP) Branch and Bound algorithms.
- Social networks use graph for finding friends, etc.
- Solving Traffic Control Problems

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

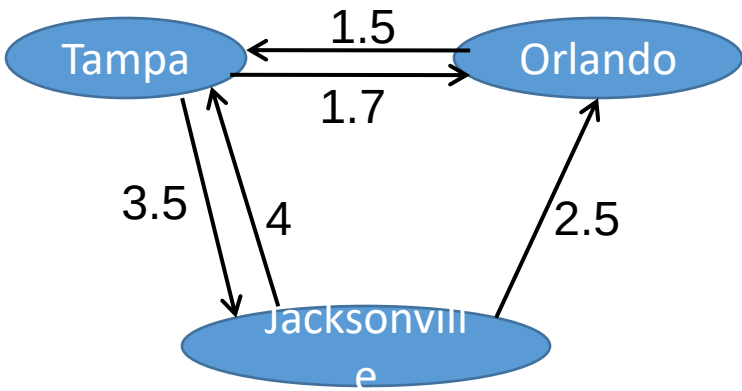
Dynamic Programming

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

- A weighted, directed graph is a collection vertices connected by weighted edges (where the weight is some real number).
  - One of the most common examples of a graph in the real world is a road map.
    - Each location is a vertex and each road connecting locations is an edge.
    - We can think of the distance traveled on a road from one location to another as the weight of that edge.

	Tampa	Orlando	Jaxville
Tampa	0	1.7	3.5
Orlando	1.5	0	∞
Jax	4	2.5	0

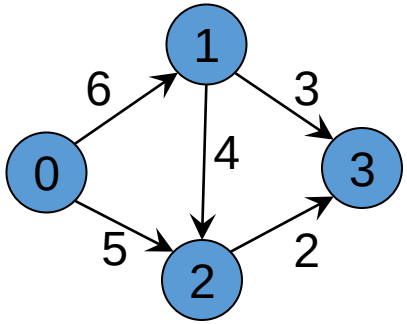


Storing a Weighted, Directed Graph

- Adjacency Matrix:
  - Let **D** be an edge-weighted graph in adjacency-matrix form
  - **D(*i*,*j*)** is the weight of edge (i, j), or ∞ if there is no such edge.
- Update matrix **D**, with the shortest path ***through immediate vertices.***

D =

	0	1	2	3
0	0	6	5	∞
1	∞	0	4	3
2	∞	∞	0	2
3	∞	∞	∞	0



Floyd-Warshall Algorithm

- Given a weighted graph, we want to know the shortest path from one vertex in the graph to another.
  - The Floyd-Warshall algorithm determines the shortest path between all pairs of vertices in a graph.
  - What is the difference between Floyd-Warshall and Dijkstra's??

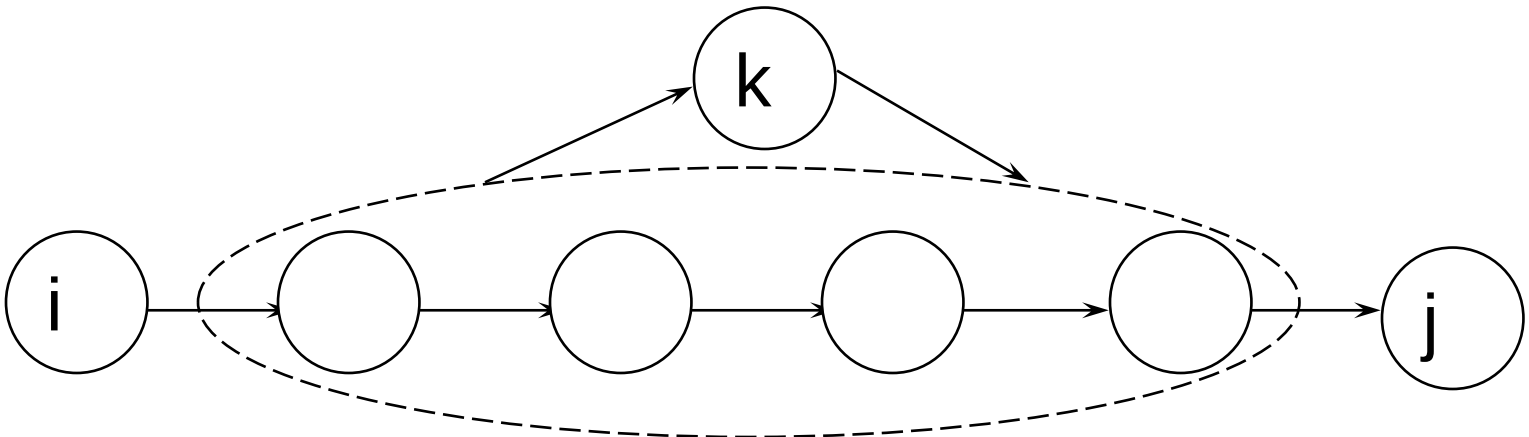
Floyd-Warshall Algorithm

- If V is the number of vertices, Dijkstra's runs in $\Theta(V^2)$
  - We could just call Dijkstra |V| times, passing a different source vertex each time.
  - $\Theta(V \times V^2) = \Theta(V^3)$
  - (Which is the same runtime as the Floyd-Warshall Algorithm)

- *BUT*, Dijkstra's doesn't work with negative-weight edges.

Floyd Warshall Algorithm

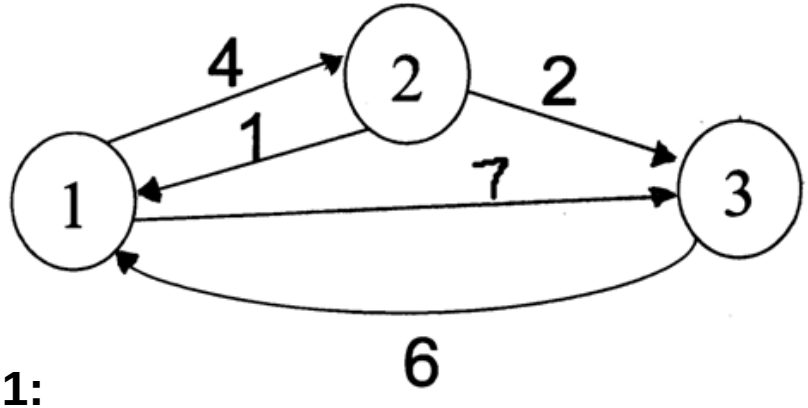
- Let's go over the premise of how Floyd-Warshall algorithm works…
  - Let the vertices in a graph be numbered from 1 … n.
  - Consider the subset {1,2,…, k} of these n vertices.
  - Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2,…,k} only.
  - There are two situations:
    1) k is an intermediate vertex on the shortest path.
    2) k is not an intermediate vertex on the shortest path.



Floyd Warshall Algorithm - Example

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$

Original weights.



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 1:

D(3,2) = D(3,1) + D(1,2)

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 2:

D(1,3) = D(1,2) + D(2,3)

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

Consider Vertex 3:

Nothing changes.

Floyd Warshall Algorithm

- Looking at this example, we can come up with the following algorithm:
  - Let D store the matrix with the initial graph edge information initially, and update D with the calculated shortest paths.

```
For k=1 to n {
  For i=1 to n {
          For j=1 to n
             D[i,j] = min(D[i,j],D[i,k]+D[k,j])
  }
}
```

  - The final D matrix will store all the shortest paths.

Floyd Warshall – Path Reconstruction

- The path matrix will store the last vertex visited on the path from i to j.
  - So path[i][j] = k means that in the shortest path from vertex i to vertex j, the LAST vertex on that path before you get to vertex j is k.

- Based on this definition, we must initialize the path matrix as follows:
  - path[i][j] = i if i!=j and there exists an edge from i to j
  - = NIL otherwise

- The reasoning is as follows:
  - If you want to reconstruct the path at this point of the algorithm when you aren't allowed to visit intermediate vertices, the previous vertex visited MUST be the source vertex i.
  - NIL is used to indicate the absence of a path.

Floyd Warshall – Path Reconstruction

- Before you run Floyd's, you initialize your distance matrix ***D*** and path matrix ***P*** to indicate the use of no immediate vertices.
  - (Thus, you are only allowed to traverse direct paths between vertices.)

- Then, at each step of Floyd's, you essentially find out whether or not using vertex k will *improve* an estimate between the distances between vertex i and vertex j.

- If it ***does improve*** the estimate here's what you need to record:

  1) record the new shortest path weight between i and j
  2) record the fact that the shortest path between i and j goes through k

Floyd Warshall – Path Reconstruction

- If it ***does improve*** the estimate here's what you need to record:
  1) record the new shortest path weight between i and j
     - **We don't need to change our path and we do not update the path matrix**
  2) record the fact that the shortest path between i and j goes through k
     - **We want to store the last vertex from the shortest path from vertex k to vertex j. *This will NOT necessarily be k, but rather, it will be path[k][j].***

This gives us the following update to our algorithm:

```
if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as
  intermediate vertex
    D[i][j] = D[i][k]+D[k][j];
    path[i][j] = path[k][j];
}
```

Path Reconstruction

- Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path.
  - Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

NIL	**3**	**4**	**5**	**1**
4	**NIL**	**4**	**2**	**1**
4	**3**	**NIL**	**2**	**1**
4	**3**	**4**	**NIL**	**1**
4	**3**	**4**	**5**	**NIL**

- Reconstruct the path from vertex1 to vertex 2:
  - First look at path[1][2] = 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2.
    - Thus, the path is now, 1…3->2.
  - Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4.
  - So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5,
  - thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1,
  - which means our path really is 1->5->4->3->2.

Transitive Closure

- Computing a transitive closure of a graph gives you complete information about which vertices are connected to which other vertices.

- Input:
  - Un-weighted graph G: $W[i][j] = 1$, if $(i,j) \in E$, $W[i][j] = 0$ otherwise.
- Output:
  - $T[i][j] = 1$, if there is a path from i to j in G, $T[i][j] = 0$ otherwise.
- Algorithm:
  - Just run Floyd-Warshall with weights 1, and make $T[i][j] = 1$, whenever $D[i][j] < \infty$.
  - More efficient: use only Boolean operators

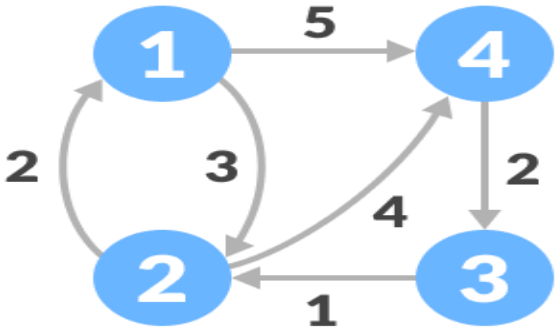
Transitive Closure

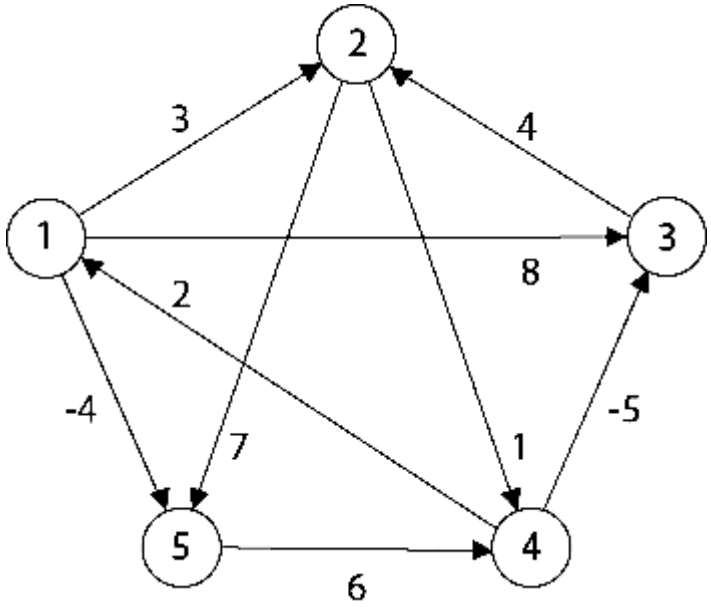
```
Transitive-Closure(W[1..n][1..n])
01 T ← W      // T(0)
02 for k ←1 to n do // compute T(k)
03    for i ←1 to n do
04       for j ←1 to n do
05          T[i][j] ← T[i][j] ∨ (T[i][k] ∧ T[k][j])
06 return T
```

- This is the SAME as the other Floyd-Warshall Algorithm, except for when we find a non-infinity estimate, we simply add an edge to the transitive closure graph.

- Every round we build off the previous paths reached.
  - After iterating through all vertices being intermediate vertices, we have tried to connect all pairs of vertices i and j through all intermediate vertices k.

How Floyd Warshall Algorithm Works?





References

- Slides adapted from Arup Guha's Computer Science II Lecture notes: http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/
- Additional material from the textbook:

  Data Structures and Algorithm Analysis in Java (Second Edition) by Mark Allen Weiss
- Additional images:

  www.wikipedia.com

  xkcd.com

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Unit-4
Backtracking

Session Learning Outcomes

- To learn about backtracking approach
- To know the applications of backtracking algorithm
- To solve problems or puzzle using backtracking approach
- To know about the concepts of N-Queens problem and Sum of subset problem
- To analyze the time complexities of N-Queens and Sum of subset problems

Motivation

- Backtracking is similar to Dynamic Programming in that it solves a problem by efficiently performing an exhaustive search over the entire set of possible options.
- Backtracking is different in that it structures the search to be able to efficiently eliminate large sub-sets of solutions that are no longer possible.
- For this reason, all backtracking algorithms will have a very similar overall structure for exhaustively searching the space of possible solutions, but the art & difficulty of the particular backtracking solution will be strategies that can be used to get rid of impossible solutions quickly.

Motivation

- Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers. All numbers will be positive integers. The solution set must not contain duplicate combinations.

Input: k = 3, n = 7

Output: [[1,2,4]]

Motivation

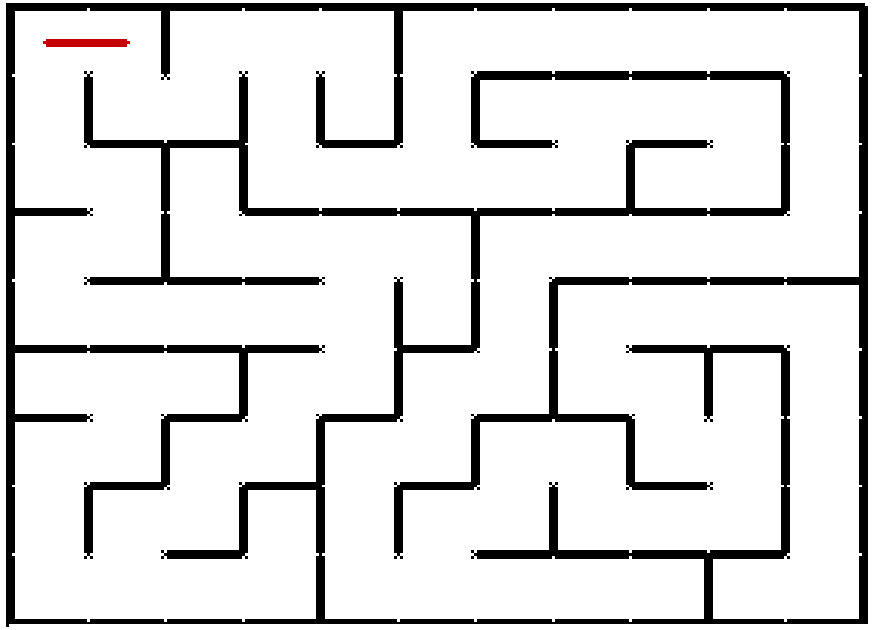
Backtracking takes solving a problem like this, by breaking it down into solving for 4 components to the algorithm.

1) Given a current sub-solution state, what are all the candidates I can choose to move to a new possible sub-solution state,
2) Decide if all possible solutions derived from the current sub-solution state will be invalid,
3) It is efficiently backtrack from any current sub-solution state to a previous state,
4) It is a particular sub-solution actually a valid solution.

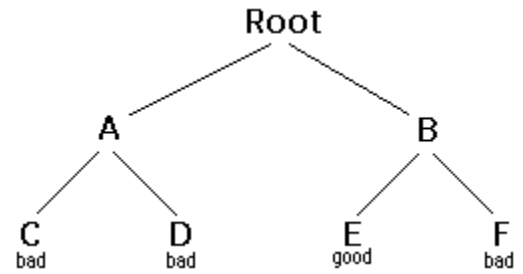
Introduction to Backtracking

- A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time.
- Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates every alternative and then chooses the best one.
- Backtracking is a systematic method to iterate through all the possible configurations of a search space.
- It is a general algorithm/technique which must be customized for each individual application.

Introduction to Backtracking



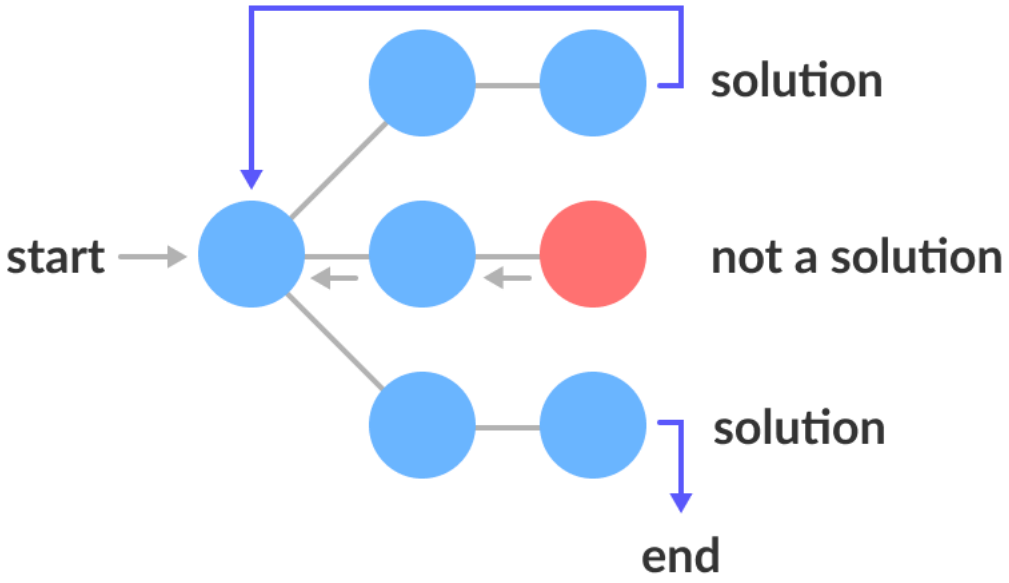
Simple Example



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

State Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.




```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

Terminologies with State Space Tree

- Goal states: These are solution states where the path from the root to the leaf defines the solution of the problem.
- Live node: A node that has been generated already but is yet to generate the children.
- E-node: A node is under consideration and is in the process of being generated.
- Dead node: A node that is already explained and cannot be considered for further searches.

Example of Backtracking Approach

- Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.
- Constraint: Girl should not be on the middle bench.
- Solution:

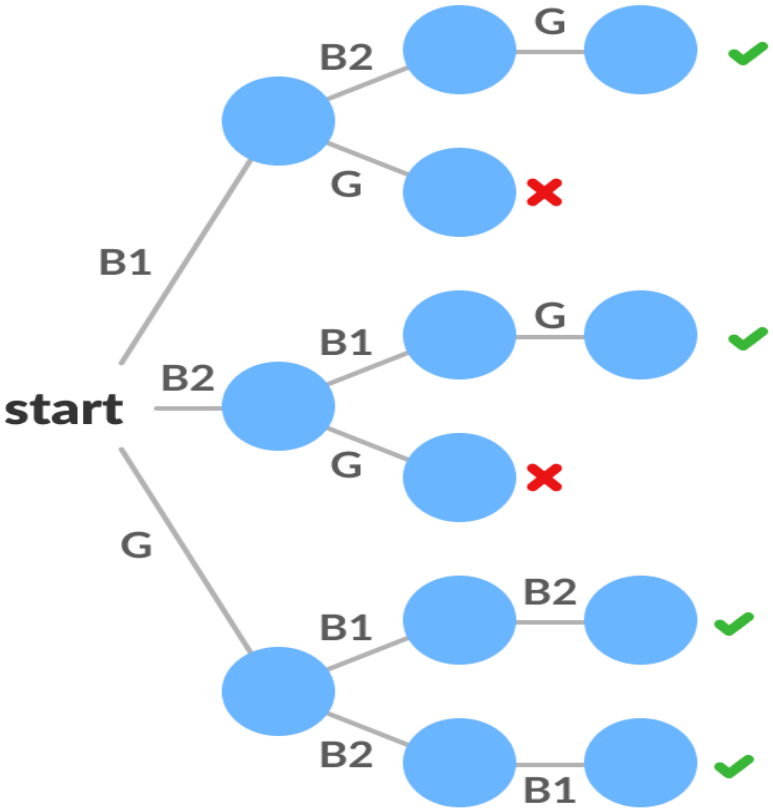
There are a total of 3! = 6 possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

B1	B2	G
B1	G	B2
B2	B1	G

B2	G	B1
G	B1	B2
G	B2	B1

Example of Backtracking Approach

- State Space Tree:



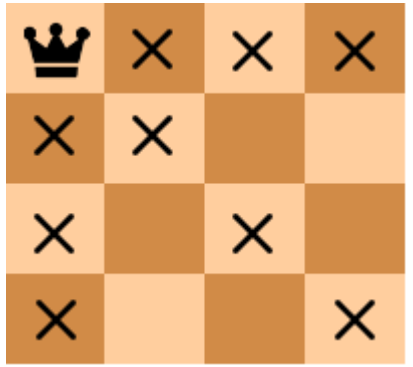
Problems solving Approach

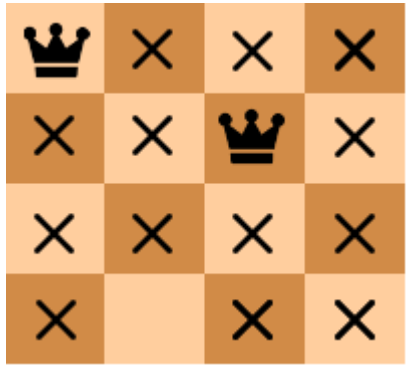
Some of the problems that can be solved using Backtracking algorithm are,

- N-Queens problem
- Sum of subset problem
- Permutations and Combinations
- Hamiltonian Circuit
- Knight Tour's problem
- Maze Solving problem
- Sudoku solving problem
- M-coloring problem
- Cryptarithmetic Puzzle
- Missionaries and Cannibals puzzle
- Magnet puzzle

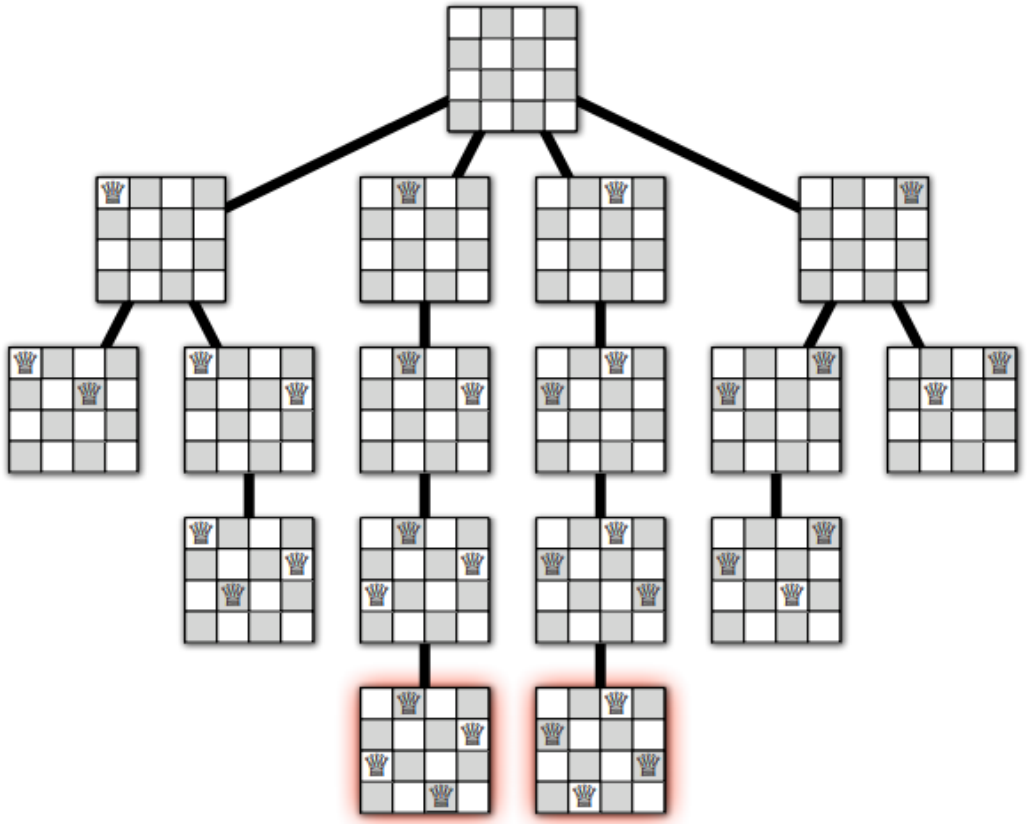
N-queens problem

- The prototypical backtracking problem is the classical n Queens Problem, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym “Schachfreund”) for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general n× n board.
- The problem is to place n queens on an n × n chessboard, so that no two queens can attack each other.
- This means that no two queens are in the same row, column, or diagonal.





State Space Tree



Algorithm

```
//Checking for the cell is attacked or not
int is_attack(int i,int j)
{
   int k,l;
   //checking if there is a queen in row or column
   for(k=0;k<N;k++)
   {
      if((board[i][k] == 1) || (board[k][j] == 1))
         return 1;
   }
```

Algorithm

```
//checking for diagonals
for(k=0;k<N;k++)
{
   for(l=0;l<N;l++)
   {
      if(((k+l) == (i+j)) || ((k-l) == (i-j)))
      {
         if(board[k][l] == 1)
            return 1;
      }
   }
}
return 0;
}
```

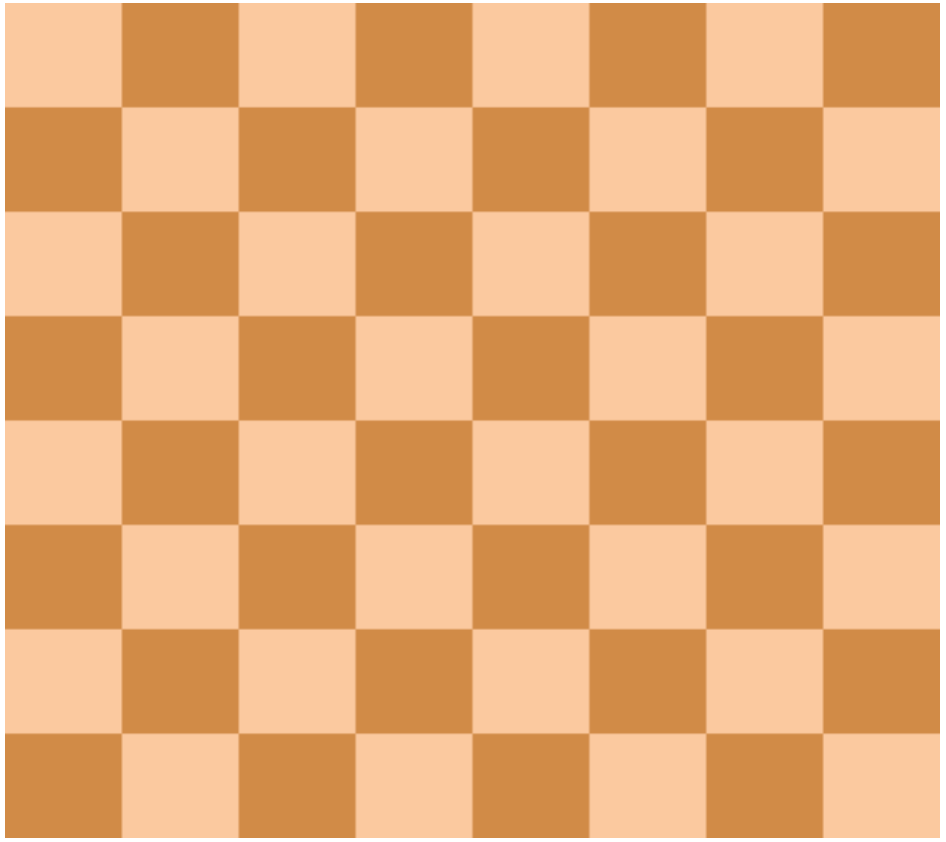
Algorithm

```
   //checking for the cell to place the queen
for(i=0;i<N;i++)
   {
      for(j=0;j<N;j++)
      {
         if((!is_attack(i,j)) && (board[i][j]!=1))
         {
            board[i][j] = 1;
            if(N_queen(n-1)==1)
            {
               return 1;
            }
            board[i][j] = 0;
         }
      }
   }
```

Algorithm

```
   //checking for the cell to place the queen
for(i=0;i<N;i++)
   {
      for(j=0;j<N;j++)
      {
         if((!is_attack(i,j)) && (board[i][j]!=1))
         {
            board[i][j] = 1;
            if(N_queen(n-1)==1)
            {
               return 1;
            }
            board[i][j] = 0;
         }
      }
   }
```

8- Queens Problem Solved



Complexity Analysis

- In general, the number of nodes that will be generated are $1+4+4^2+....+4^n= 4^{n+1}-1/4-1= 4^{n+1}-1/3$
- But the constraint that no two queens in an attacking position reduces the calculation to

  (For four queens) 1+4*3+4*3*2+4*3*2*1 promising nodes

  In general,

  n+n(n-1)+n(n-1)(n-2)+…….+1 = n! promising nodes are possible.
- The best, average and worst case time complexity of this algorithm is O(N!) where N is number of queens.

Sum of Subsets Problem

- Subset sum problem is the problem of finding a subset such that the sum of elements equal a given number.
- The backtracking approach generates all permutations in the worst case but in general, performs better than the recursive approach towards subset sum problem.
- A subset A of n positive integers and a value **sum** is given, find whether or not there exists any subset of the given set, the sum of whose elements is equal to the given value of sum.
- Given the following set of positive numbers:

  { 2, 9, 10, 1, 99, 3}

  For 4, Ans: {1, 3}

  For 5, Ans: {2, 3}

Algorithm

1.Start with an empty set

2.Add the next element from the list to the set

3.If the subset is having sum M, then stop with that subset as solution.

4.If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.

5.If the subset is feasible (sum of subset < M) then go to step 2.

6.If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution

Pseudocode

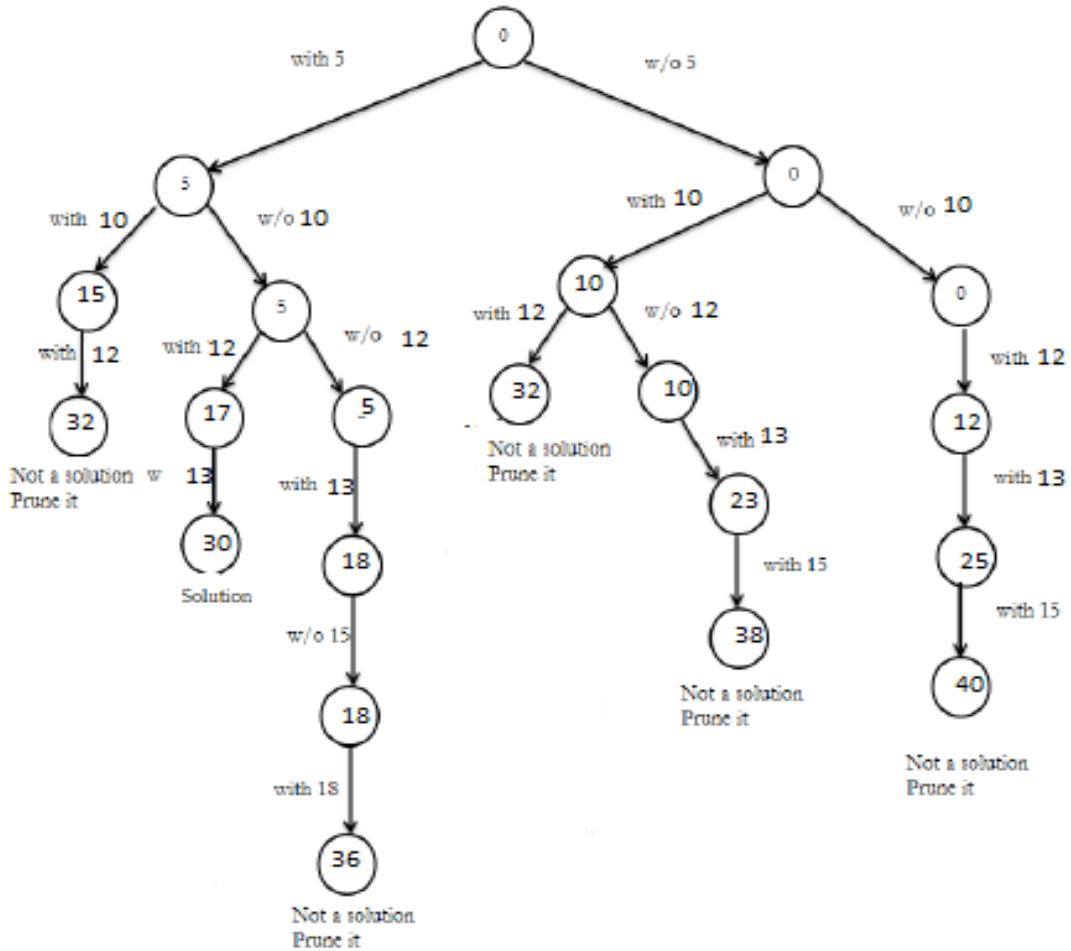
```
void subset_sum(int list[], int sum, int starting_index, int target_sum)
{
      if( target_sum == sum )
      {
         subset_count++;
         if(starting_index < list.length)
            subset_sum(list, sum - list[starting_index-1], starting_index, target_sum);
      }
      else
      {
         for( int i = starting_index; i < list.length; i++ )
         {
            subset_sum(list, sum + list[i], i + 1, target_sum);
         }
      }
   }
```

Example

- Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18}

Initially subset = {}	Sum = 0	Description
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

State Space Tree



Analysis

- The time complexity of solving Subset sum problem using backtracking approach is O(2^N) where N is the given number elements in the set.

Problems to solve

- How many solutions are possible for 4-queens problem?
- Give one solution of 8-queens problems?
- Implement 4-queens problem using any languages
- Write a recursive and non-recursive procedure for backtracking
- Find the sum of subsets for the following set of integers.

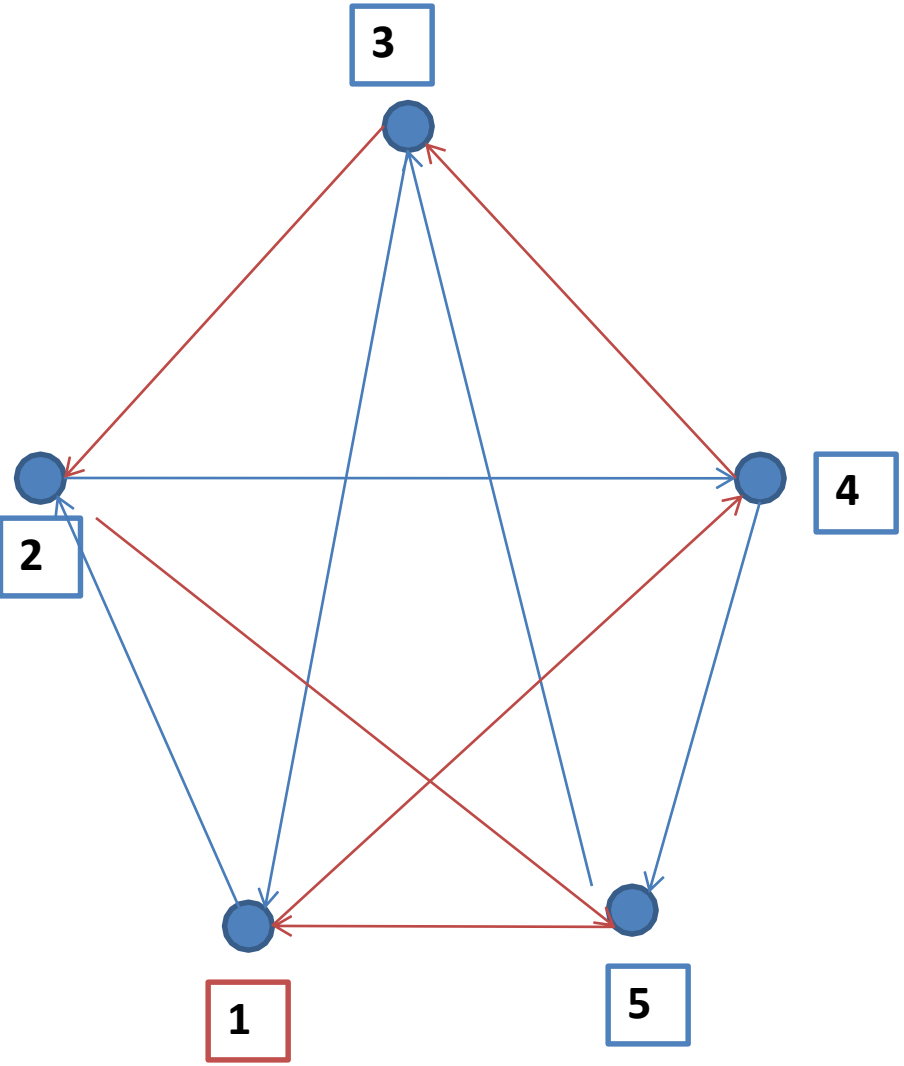
  [5 10 25 50 100] for W=75

References

1. *Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014*
2. *Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2006*
3. *S. Sridhar, Design and Analysis of Algorithms, Oxford University Press, 2015*
4. *https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/*

Travelling Salesman Problem

Travelling salesman Problem-Definition



- Let us look at a situation that there are 5 cities, Which are represented as NODES
- There is a Person at NODE-1

- This **PERSON HAS TO REACH EACH NODES ONE AND ONLY ONCE AND COME BACK TO ORIGINAL (STARTING)POSITION.**
- This **process has to occur with minimum cost or minimum distance travelled.**

- Note that starting point can start with any Node. For Example:

1-5-2-3-4-1

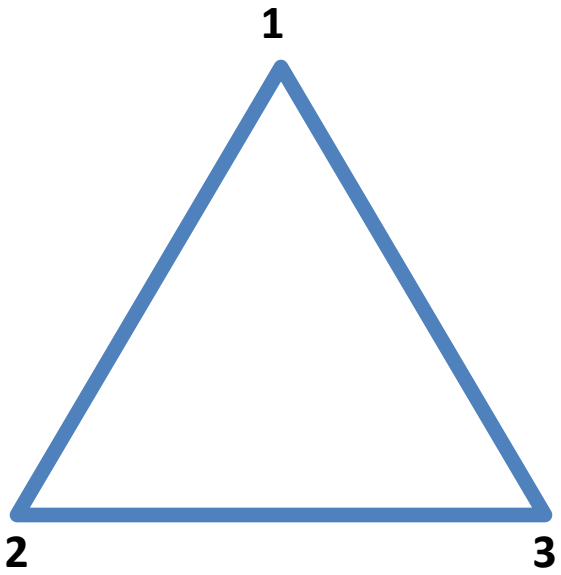
2-3-4-1-5-2

Travelling salesman Problem-Definition

- If there are ‘n’ nodes there are (n-1)! Feasible solutions
- From these (n-1)! Feasible solutions we have to find OPTIMAL SOLUTION.
- This can be related to GRAPH THEORY.
- Graph is a collection of Nodes and Arcs(Edges).

Travelling salesman Problem-Definition

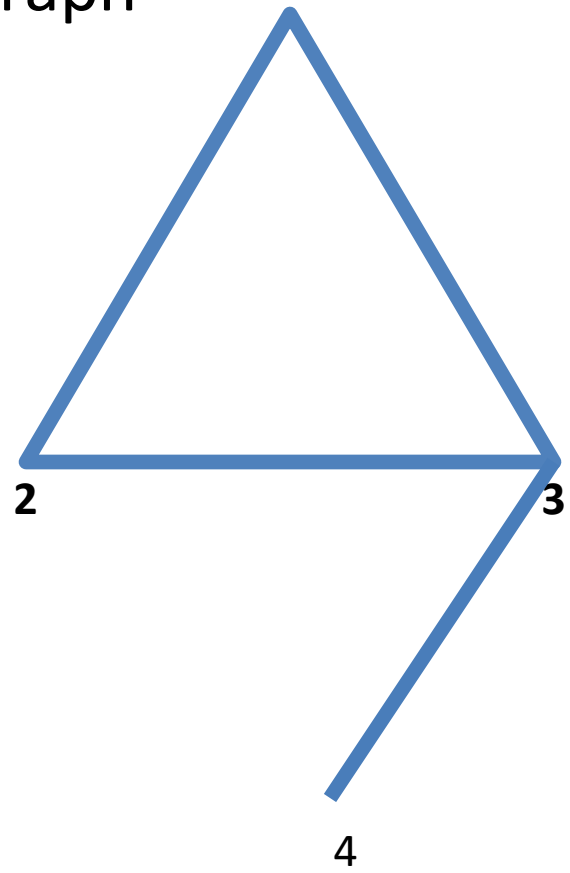
- Let us say there are Nodes Connected as shown
- We can find a Sub graph as 1-3-2-1.Hence this **GRAPH IS HAMILTONIAN**



Travelling salesman Problem-Definition

- But let us consider this graph
- We can go to

  1-3-4-3-2-1



But we are reaching 3 again to make a cycle.

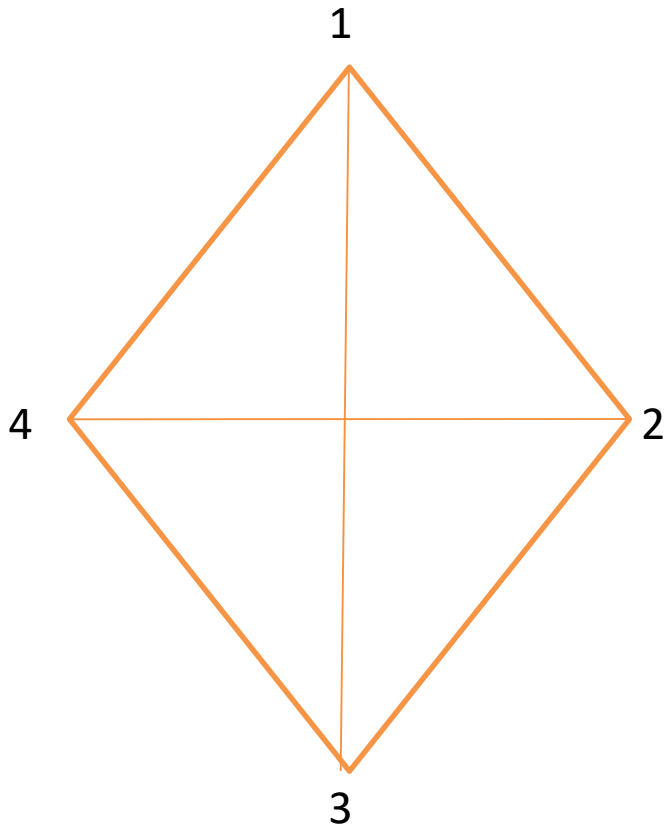
HENCE THIS GRAPH IS NOT HAMILTONIAN

HAMILTONIAN GRAPHS

- The Given Graph is Hamiltonian
- If a graph is Hamiltonian, it may have more than one Hamiltonian Circuits.
- For eg:

1-4-2-3-1

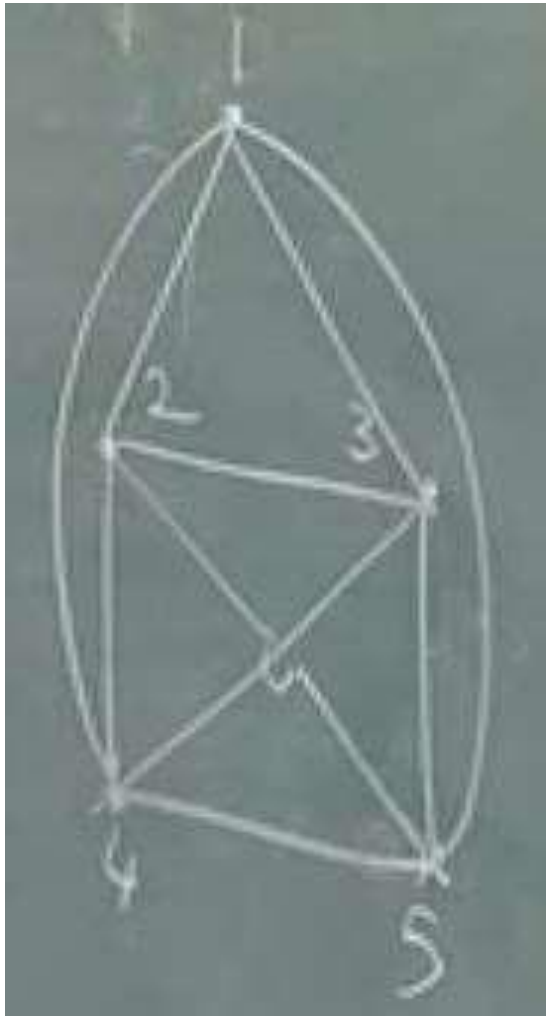
1-2-3-4-1 etc.,



Hamiltonian Graphs And Travelling Salesman Problem

- Graphs Which are Completely Connected i.e., if we have Graphs with every vertex connected to every other vertex, then Clearly That graph is HAMILTONIAN.

- So Travelling Salesman Problem is nothing but finding out LEAST COST HAMILTONIAN CIRCUIT



Travelling salesman Problem Example

	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

Here Every Node is connected to every other Node.

But the cost of reaching the same node from that node is Nil. So only a DASH is put over there.

Since Every Node is connected to every other Node various Hamiltonian Circuits are Possible.

Travelling salesman Problem Example

	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

We can have various Feasible Solutions.

For Example
1-2-4-5-3-1
2-5-1-4-3-2
Etc…

But From these Feasible Solutions We want to find the optimal Solution.

We should not have SUBTOURS.

It should comprise of TOURS.

Travelling salesman Problem Example-Formulations

- X_{ij} = 1,if person moves IMMEDIATELY from I to j.
- Objective Function is to minimize the total distance travelled which is given by

$$\sum\sum C_{ij}\, X_{ij}$$

Where C_{ij} is given by Cost incurred or Distance Travelled

$$\text{For } j=1 \text{ to } n,\ \sum X_{ij}=1, \qquad \forall\ i$$

$$\text{For } i=1 \text{ to } n,\ \sum X_{ij}=1, \qquad \forall j$$

$$X_{ij}=0 \text{ or } 1$$

Sub Tour Elimination Constraints

- We can have Sub tours of length n-1
- We eliminate sub tour of length 1 By making Cost to travel from j to j as infinity.

$$C_{jj}=\infty$$

- To eliminate Sub tour of Length 2 we have

$$X_{ij}+X_{ji}<=1$$

- To eliminate Sub tour of Length 3 we have

$$X_{ij}+X_{jk}+X_{ki}<=2$$

- If there are n nodes Then we have the following constraints
- nc_2 for length 2
- nc_3 for length 3
- ……
- nc_{n-1} for length n-1

Travelling salesman Problem Example

Sub tour elimination

	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

TSP - SOLUTIONS

- Branch and Bound Algorithm
- Heuristic Techniques

Travelling salesman Problem Example

Row Minimum

	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

Total Minimum Distance
= Sum of Row Minima

Here Total

Minimum Distance =31.

Lower Bound=31 that a person should surely travel. Our cost of optimal Solution should be surely greater than or equal to 31

Travelling salesman Problem Example

Column Minimum

	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

Total Minimum Distance =Sum of Column Minima

Here Total Minimum Distance also =31

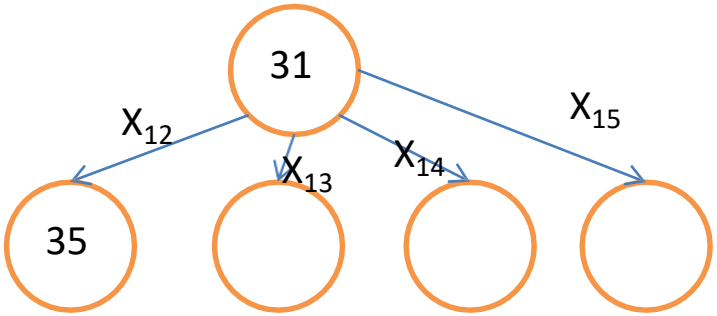
Hence the Problem

Matrices is **Symmetric.**

TSP USUALLY SATISFIES

1. SQUARE
2. SYMMETRIC

Branch and Bound-Step

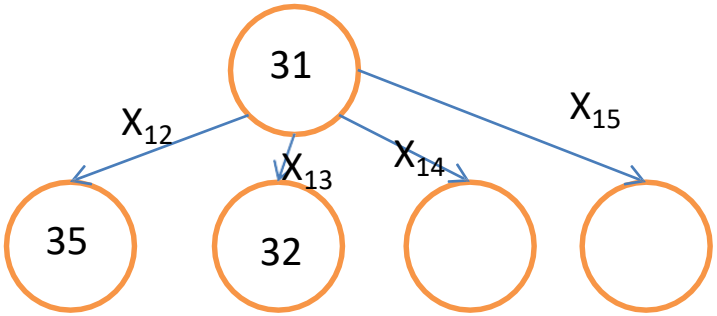


	1	3	4	5
2	-	10	5	6
3	8	-	8	9
4	9	8	-	6
5	7	9	6	-

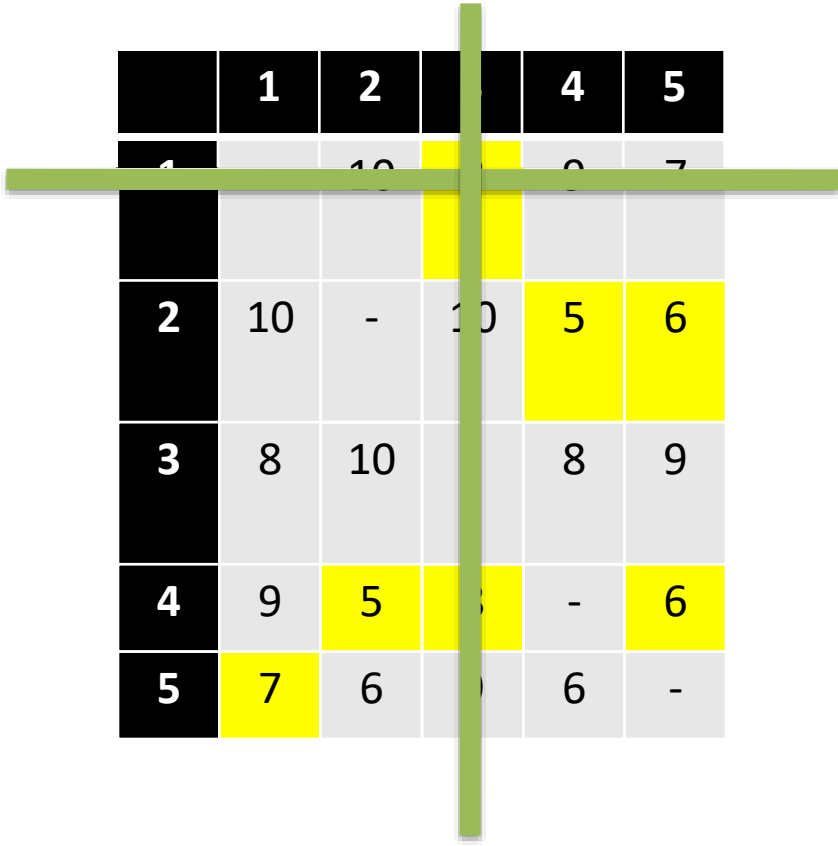
	1		3	4	5
2	10		10	5	6
3	8		-	8	9
4	9		8	-	6
5	7		9	6	-

For X_{12} 10+5+8+6+6=35

Branch and Bound-Step



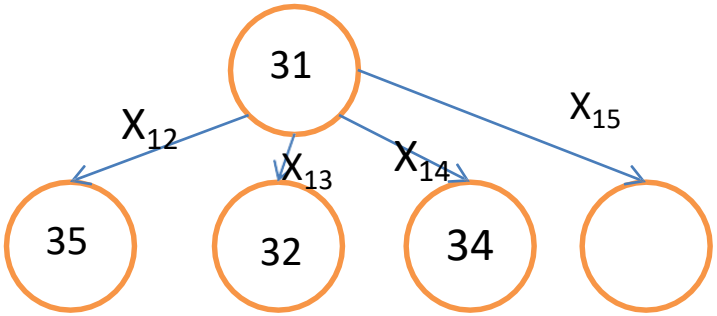
	1	2	4	5
2	10	-	5	6
3	-	10	8	9
4	9	5	-	6
5	7	6	6	-



	1	2		4	5
2	10	-	10	5	6
3	8	10		8	9
4	9	5		-	6
5	7	6		6	-

For X_{13} 8+5+8+5+6=32

Branch and Bound-Step

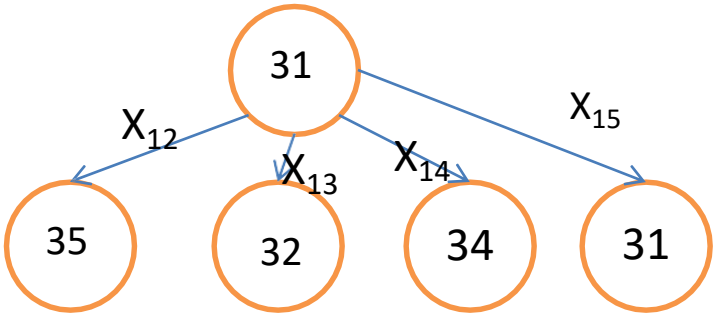


	1	2	3	5
2	10	-	10	6
3	8	10	-	9
4	-	5	8	6
5	7	6	9	-

	1	2	3	4	5
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

For X_{14} 9+6+8+5+6=34

Branch and Bound-Step

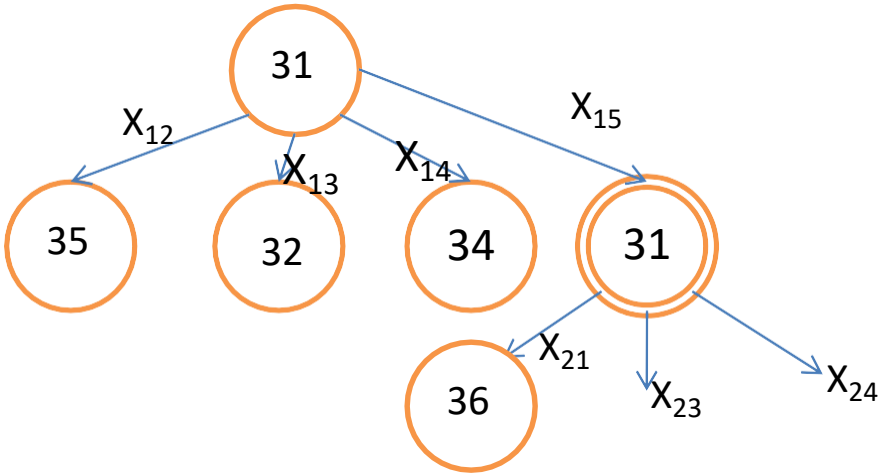


	1	2	3	4
2	10	-	10	5
3	8	10	-	8
4	9	5	8	-
5	-	6	9	6

	1	2	3	4	5
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

For X_{15} 7+5+8+5+6=31

Branch and Bound-Step

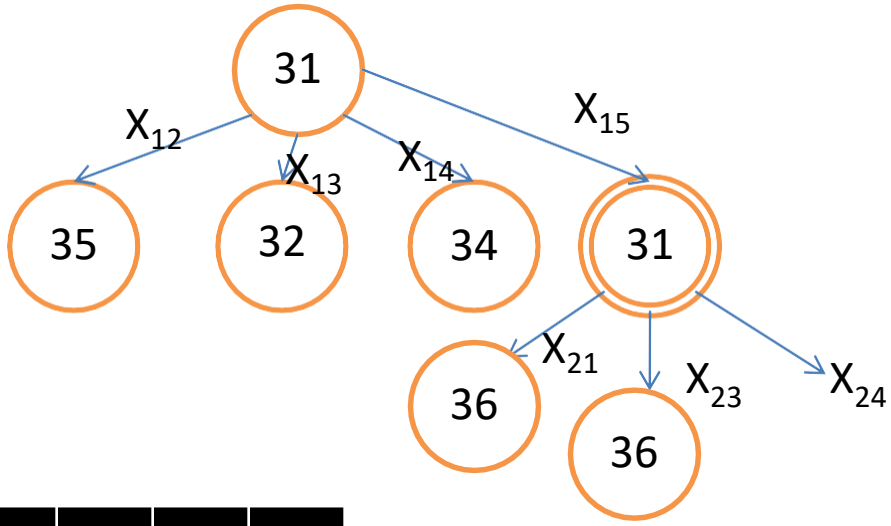


	2	3	4
3	10	-	8
4	5	8	-
5	-	9	6

		2	3	4	
3	8	10	-	8	
4	9	5	8	-	
5	7	6	9	6	

For X_{15} And X_{21}
7+10+8+5+6=36

Branch and Bound-Step

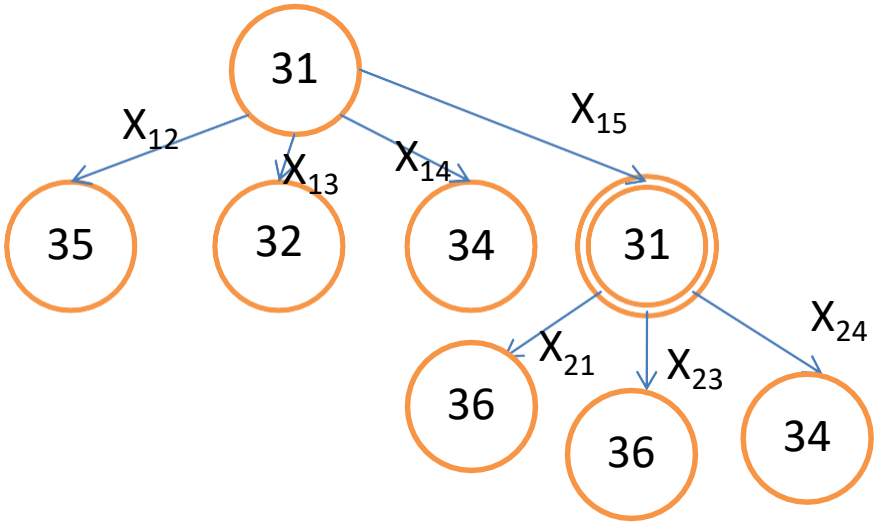


	1	2	4
3	8 10	10	8
4	9	5	-
5	7	6	6

	1	2	4
3	8	10	8
4	9	5	-
5	7	6	6

For X_{15}
And X_{23}
7+10+8+5+6=36

Branch and Bound-Step

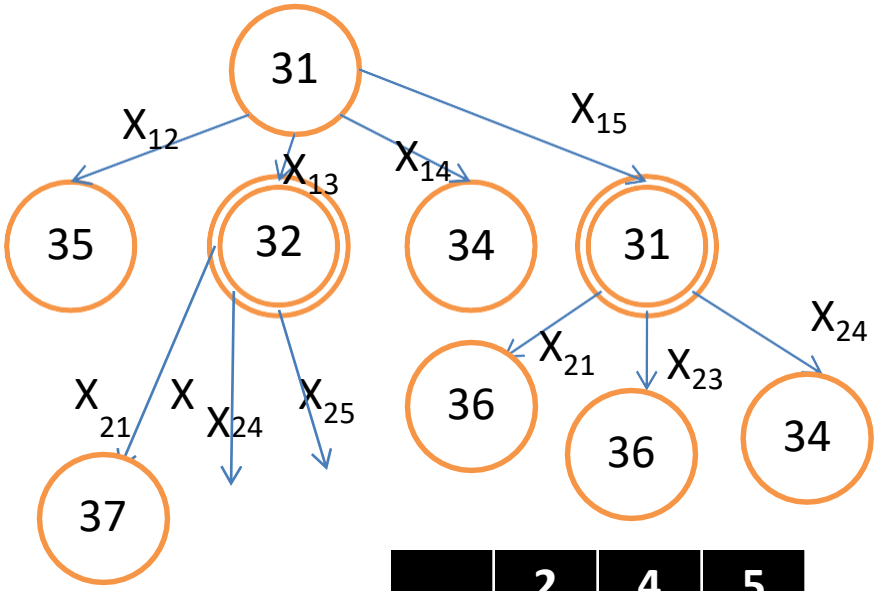


	1	2	3
3	8	10	-
4	9	-	8
5	7	6	9

	1	2	3		
3	8	10	-		
4	9	5	8		
5	7	6	9		

For X_{15} And X_{24}
7+5+8+8+6=34

Branch and Bound-Step

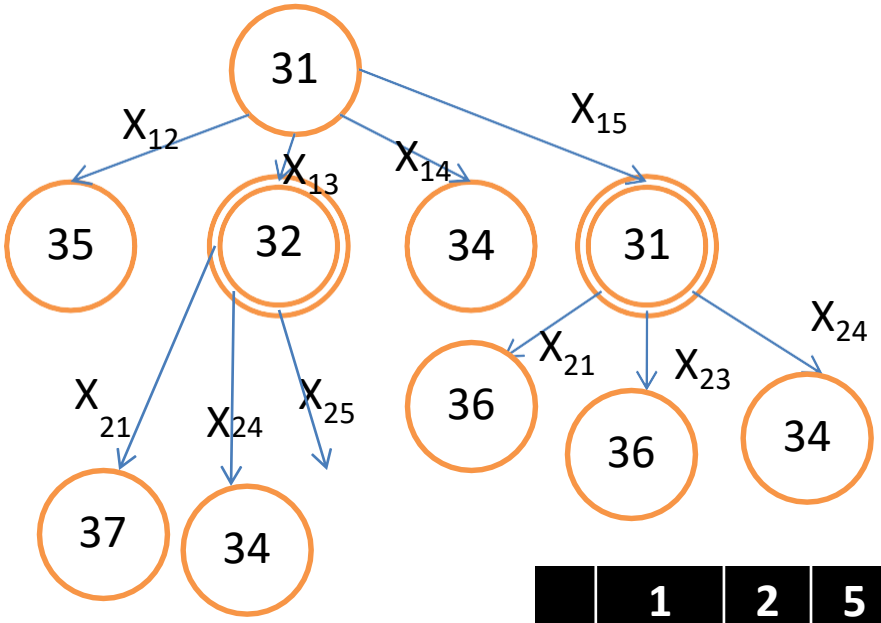


	2	4	5
3	-	8	9
4	5	-	6
5	6	6	-

				4	5
1		10		9	7
3		10		8	9
4		5		-	6
5		6		6	-

For X_{13} And X_{21}
8+10+8+5+6=37

Branch and Bound-Step

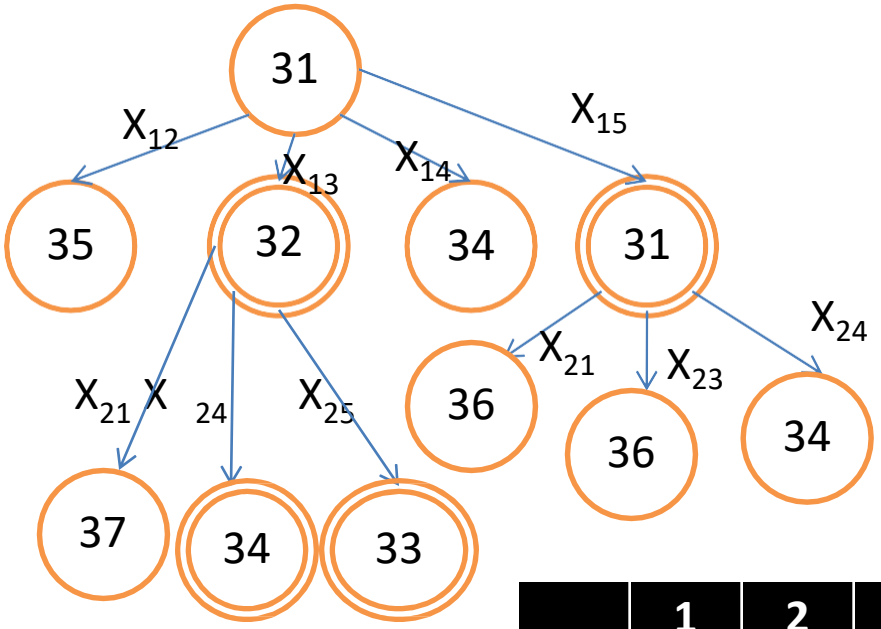


	1	2	5
3	8	10	9
4	9	-	6
5	7	6	-

	1	2			5
1	-	10			7
3	8	10			9
4	9	5			6
5	7	6			-

For X_{13} And X_{24}
8+5+9+6+6=34

Branch and Bound-Steps

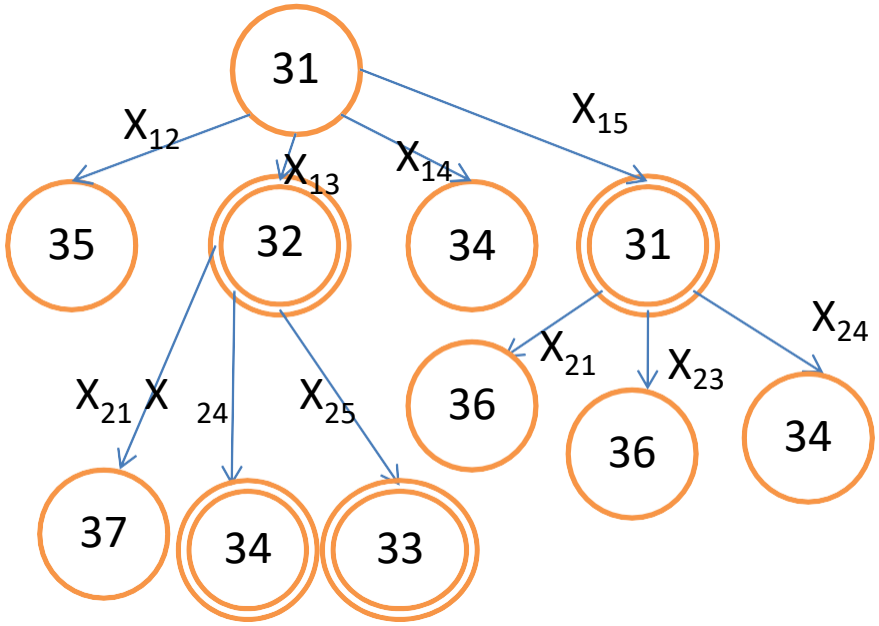


	1	2	4
3	8	10	8
4	9	5	-
5	7	-	6

	1	2		4	
3	8	10		8	
4	9	5		-	
5	7	6		6	

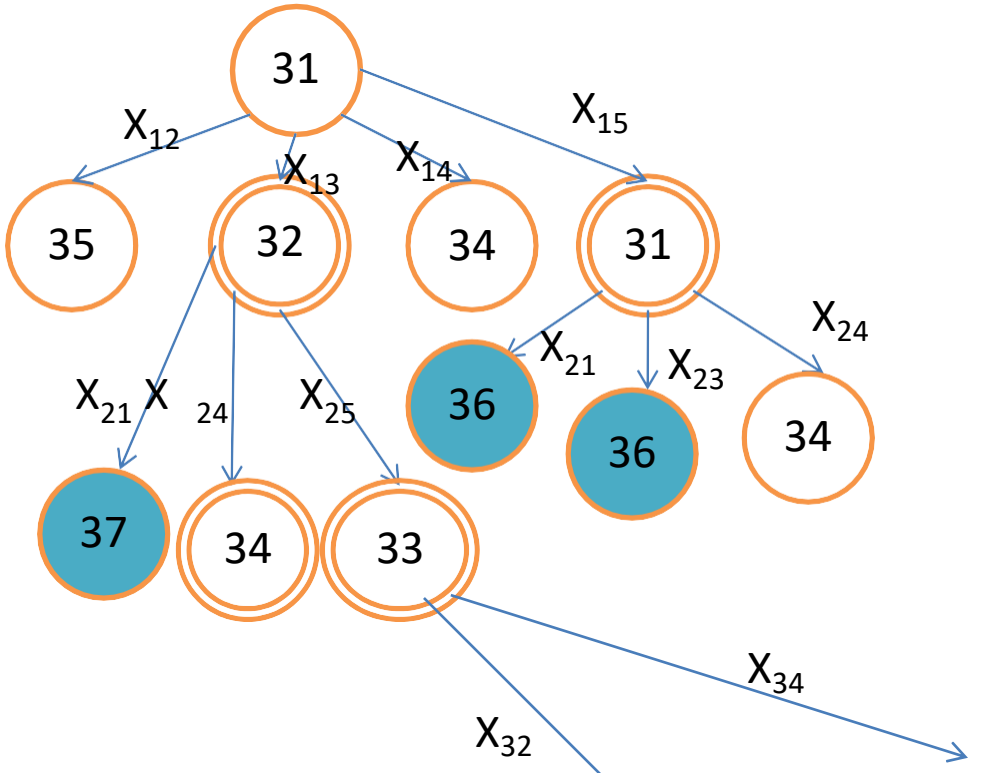
For X_{13} And X_{25}
8+6+8+5+6=33

Branch and Bound-Steps



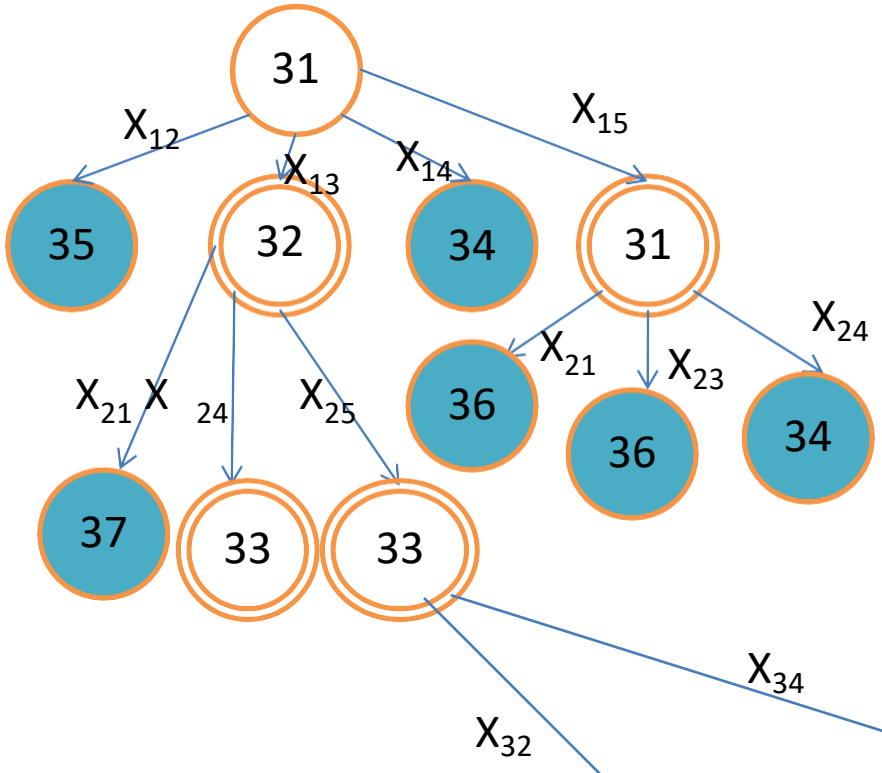
	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

Branch and Bound-Steps



	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

Branch and Bound-Steps



	1	2	3	4	5
1	-	10	8	9	7
2	10	-	10	5	6
3	8	10	-	8	9
4	9	5	8	-	6
5	7	6	9	6	-

1-3-4-2-5-1
Distance=8+8+5+6+7=34
This is the Optimal Solution.
This is same as
1-5-2-4-3-1

Example Problems 1

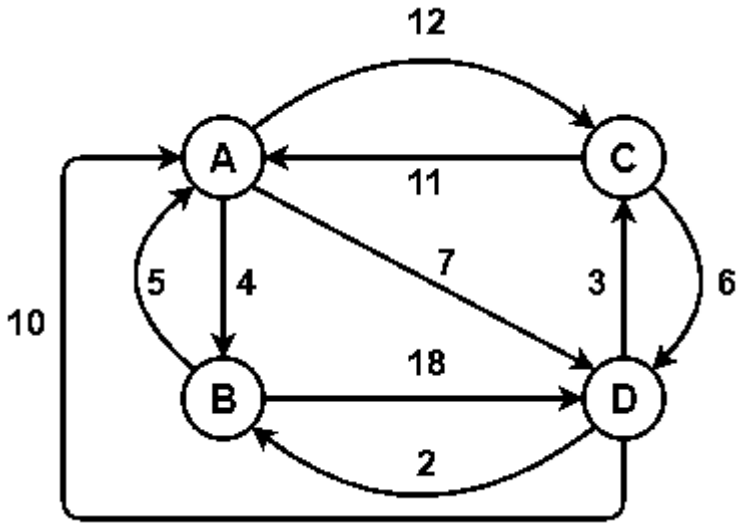
A travelling salesman, named Mathew plans to visit five cities 1, 2, 3, 4 & 5. The travel time (in hours) between these cities is shown below:

	To				
From	1	2	3	4	5
1	∞	5	8	4	5
2	5	∞	7	4	5
3	8	7	∞	8	6
4	4	4	8	∞	8
5	5	5	6	8	∞

How should Mr. Mathew schedule his touring plan in order to **minimize** the total travel **time**, if he visits each city once a week?

Example Problems 2

The following graph shows a set of cities and distance between every pair of cities. if salesman starting city is A, then find the TSP tour in the graph.



References

Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd Ed., The MIT Press Cambridge, 2014.

S. Sridhar, Design and Analysis of Algorithms, Oxford University Press, 2015

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/

https://nptel.ac.in/courses/112/106/112106131/

BRANCH AND BOUND - knapsack

Session Learning Outcome-SLO

At the end of this session student would have gained sufficient knowledge regarding design technique

– Branch and bound and one of its application in optimization (Knapsack problem)

- Student should be trained to identify the type of knapsack problem that requires branch and bound to get an optimized solution.

Motivation behind using this technique comes with the concept of expecting maximum profit in any situation and as well as minimizing the cost or resource in any situation.

0/1 Knapsack Problem

Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively. Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W. We have ‘n’ items with value v1 , v2 . . . vn and weight of the corresponding items is w1 , w2 . . . Wn . Max capacity is W . We can either choose or not choose an item. We have x1 , x2 . . . xn. Here xi = { 1 , 0 }. xi = 1 , item chosen xi = 0 , item not chosen

Different approaches of this problem :

- ✓ Dynamic programming
- ✓ Brute force
- ✓ Backtracking
- ✓ Branch and bound

What is branch and bound ?

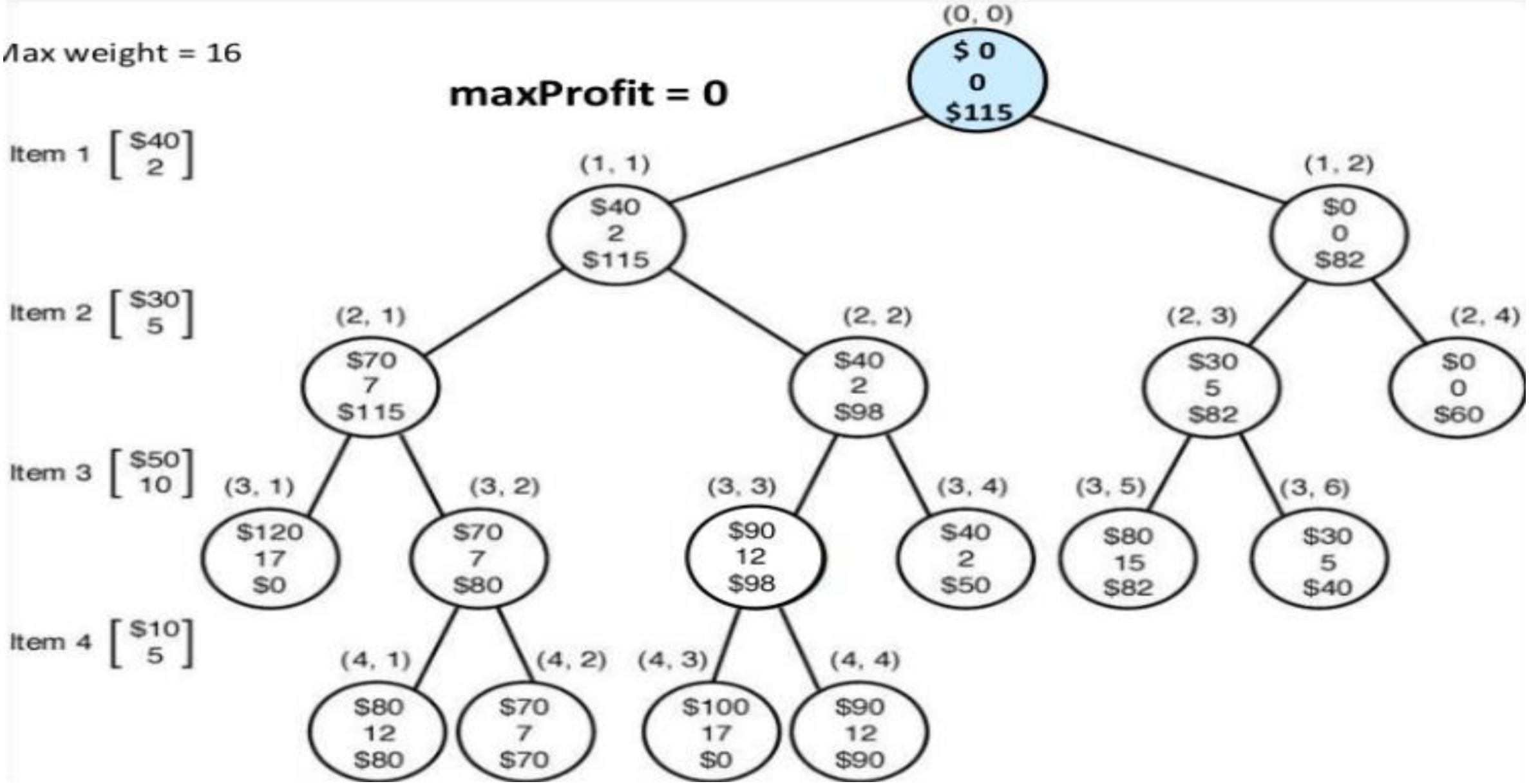
- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems.
- These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- Branch and Bound solve these problems relatively quickly.
- Combinatorial optimization problem is an optimization problem, where an optimal solution has to be identified from a finite set of solutions

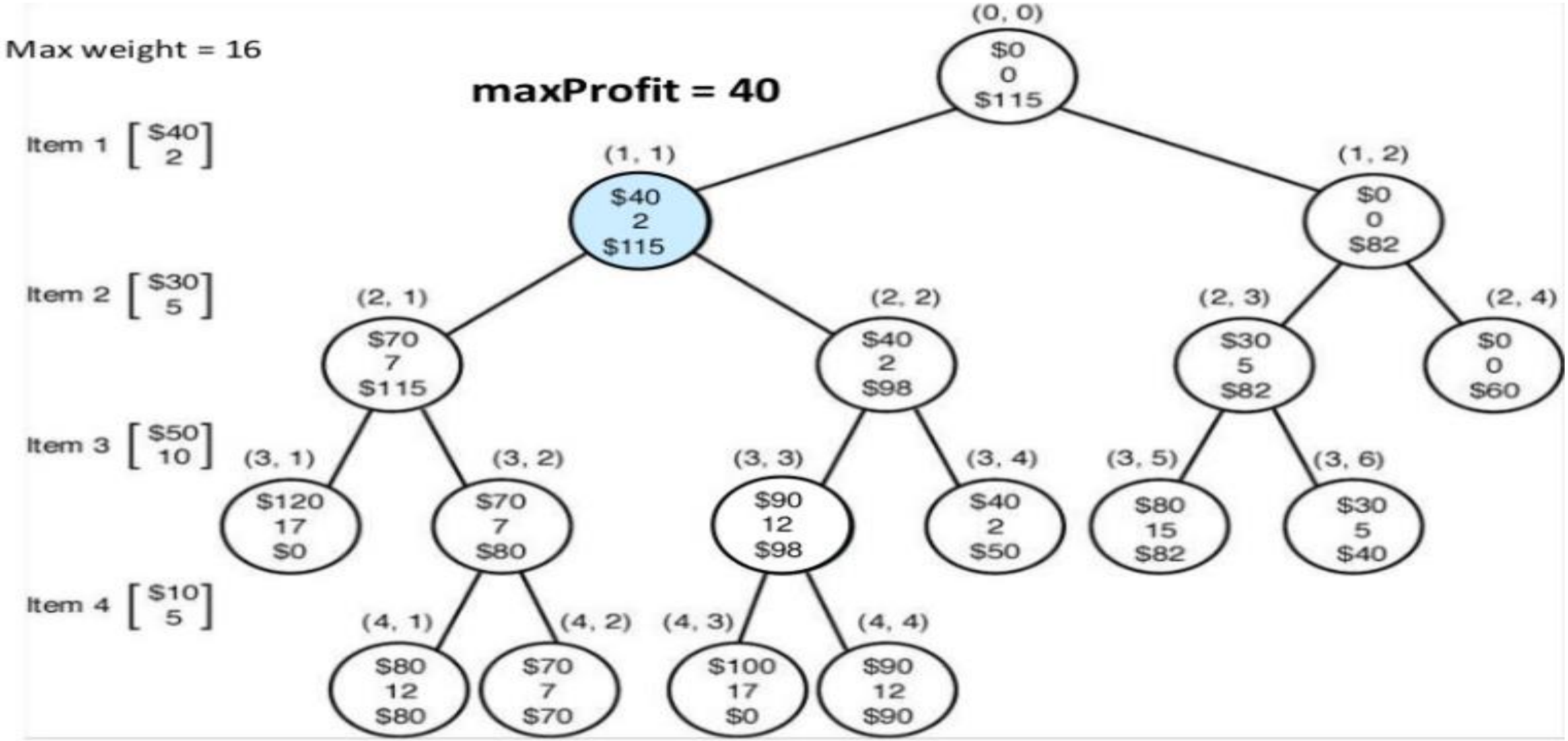
ALGORITHM

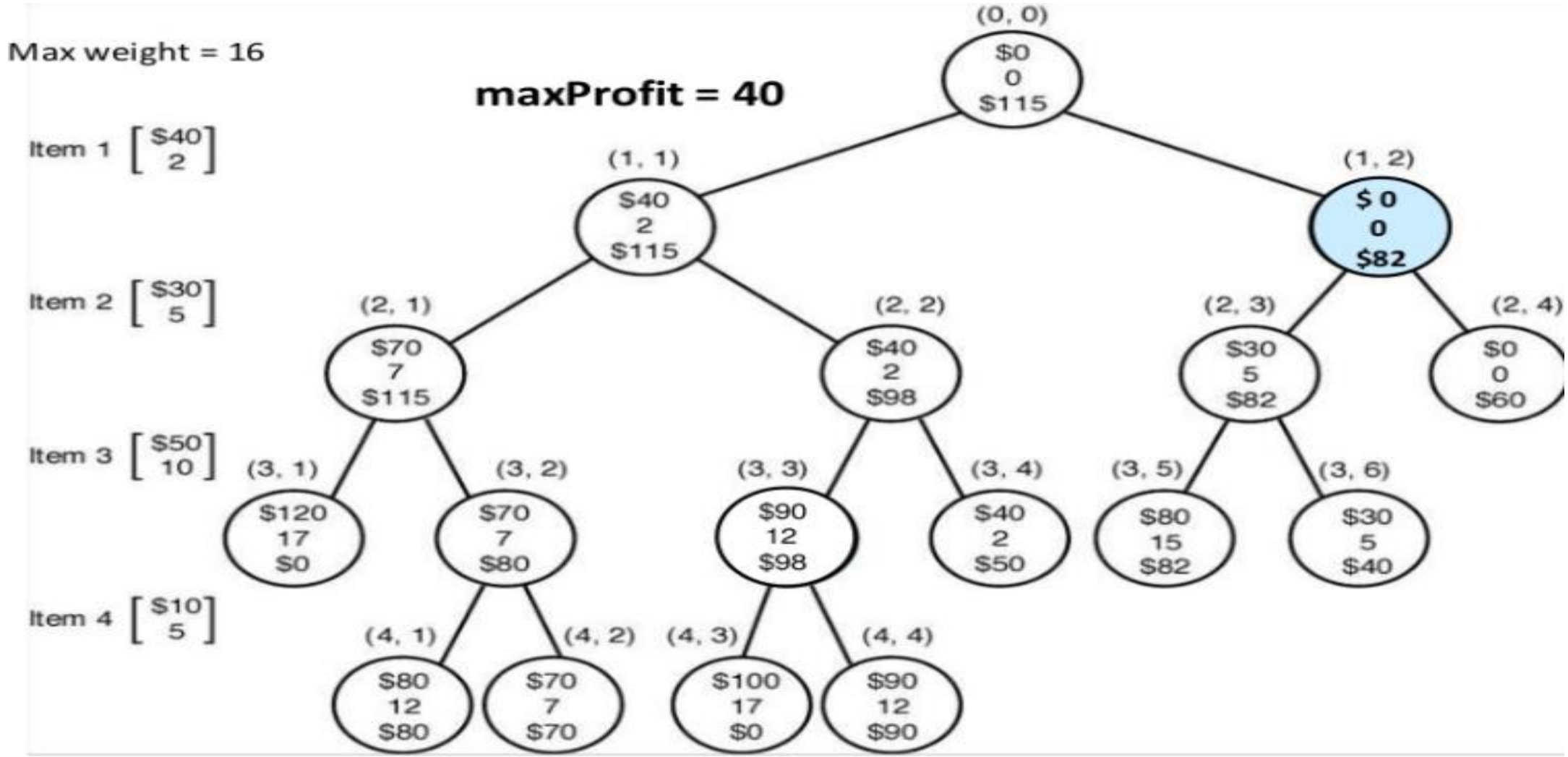
- Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
- Initialize maximum profit, maxProfit = 0
- Create an empty queue, Q.
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.

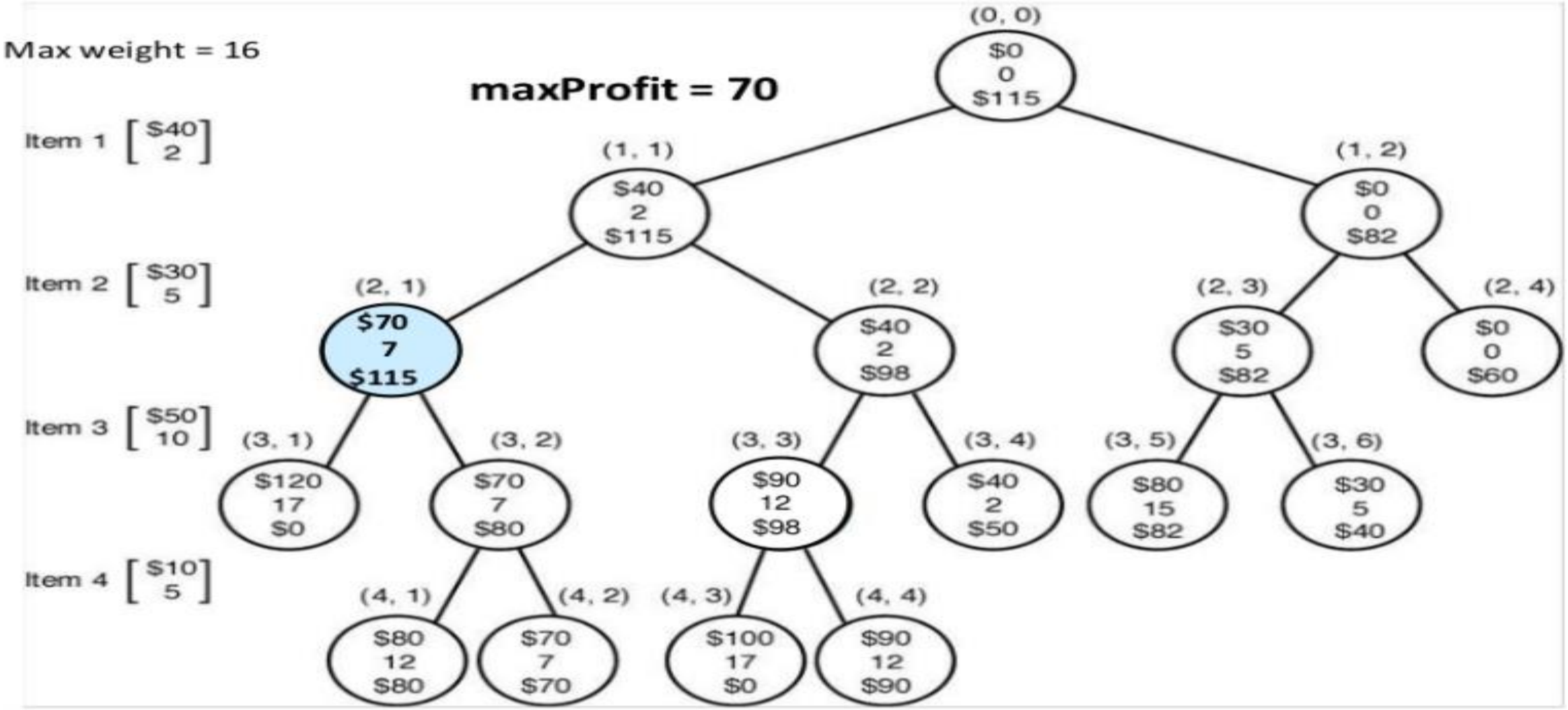
Do following while Q is not empty

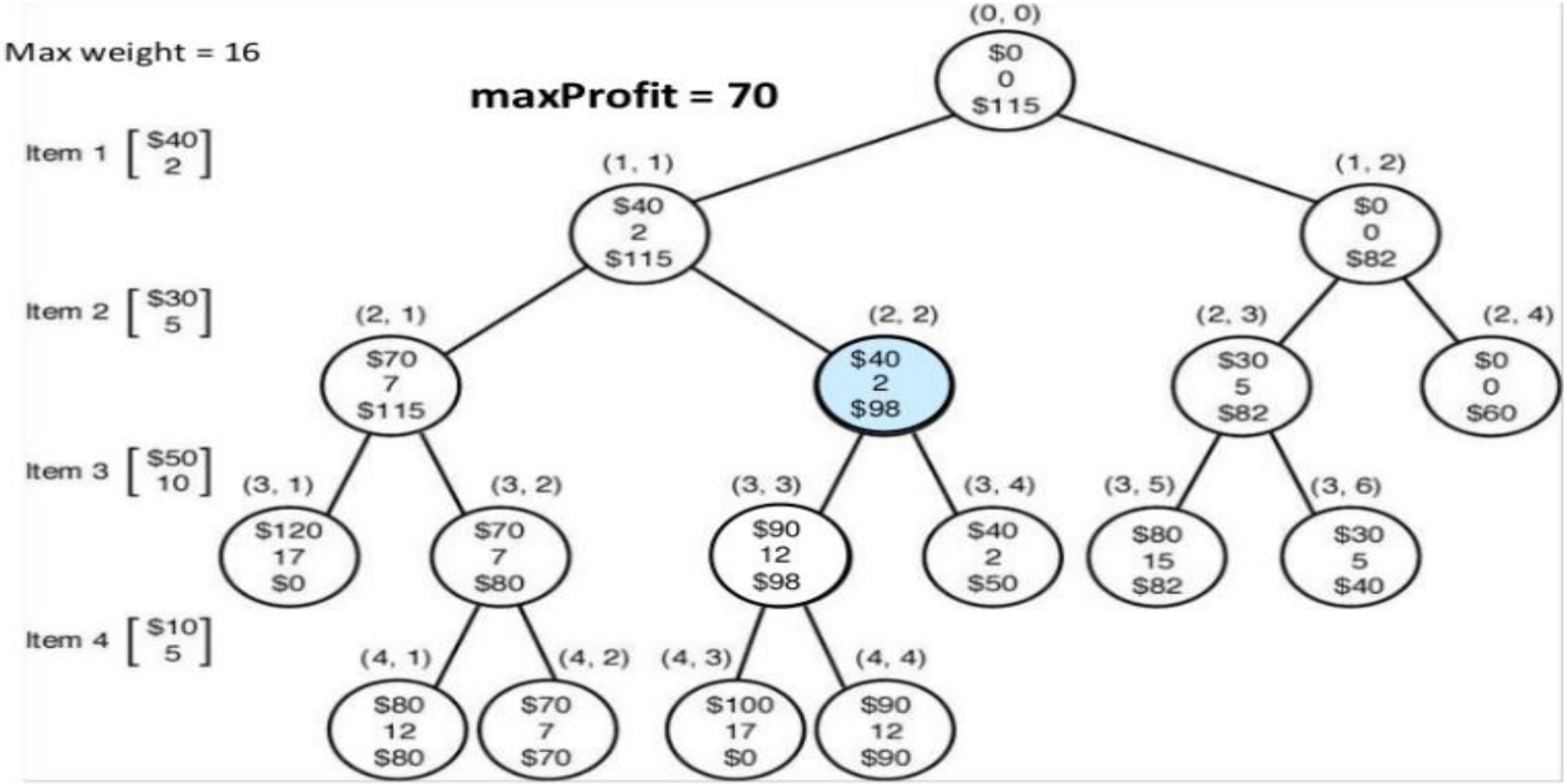
- Extract an item from Q. Let the extracted item be u. Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

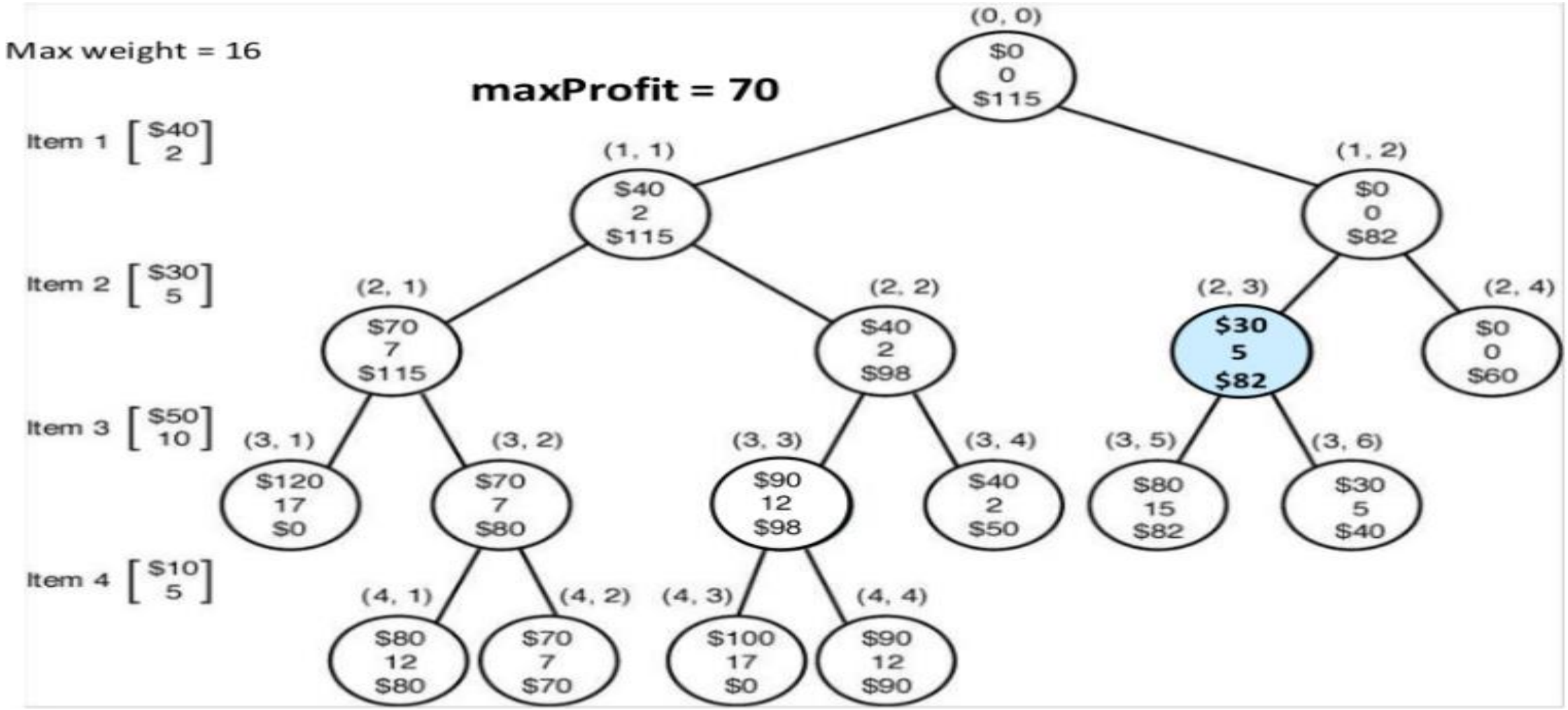

Max weight = 16
maxProfit = 0
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
(0, 0)
$ 0
0
$115
(1, 1)
$40
2
$115
(1, 2)
$0
0
$82
(2, 1)
$70
7
$115
(2, 2)
$40
2
$98
(2, 3)
$30
5
$82
(2, 4)
$0
0
$60
(3, 1)
$120
17
$0
(3, 2)
$70
7
$80
(3, 3)
$90
12
$98
(3, 4)
$40
2
$50
(3, 5)
$80
15
$82
(3, 6)
$30
5
$40
(4, 1)
$80
12
$80
(4, 2)
$70
7
$70
(4, 3)
$100
17
$0
(4, 4)
$90
12
$90

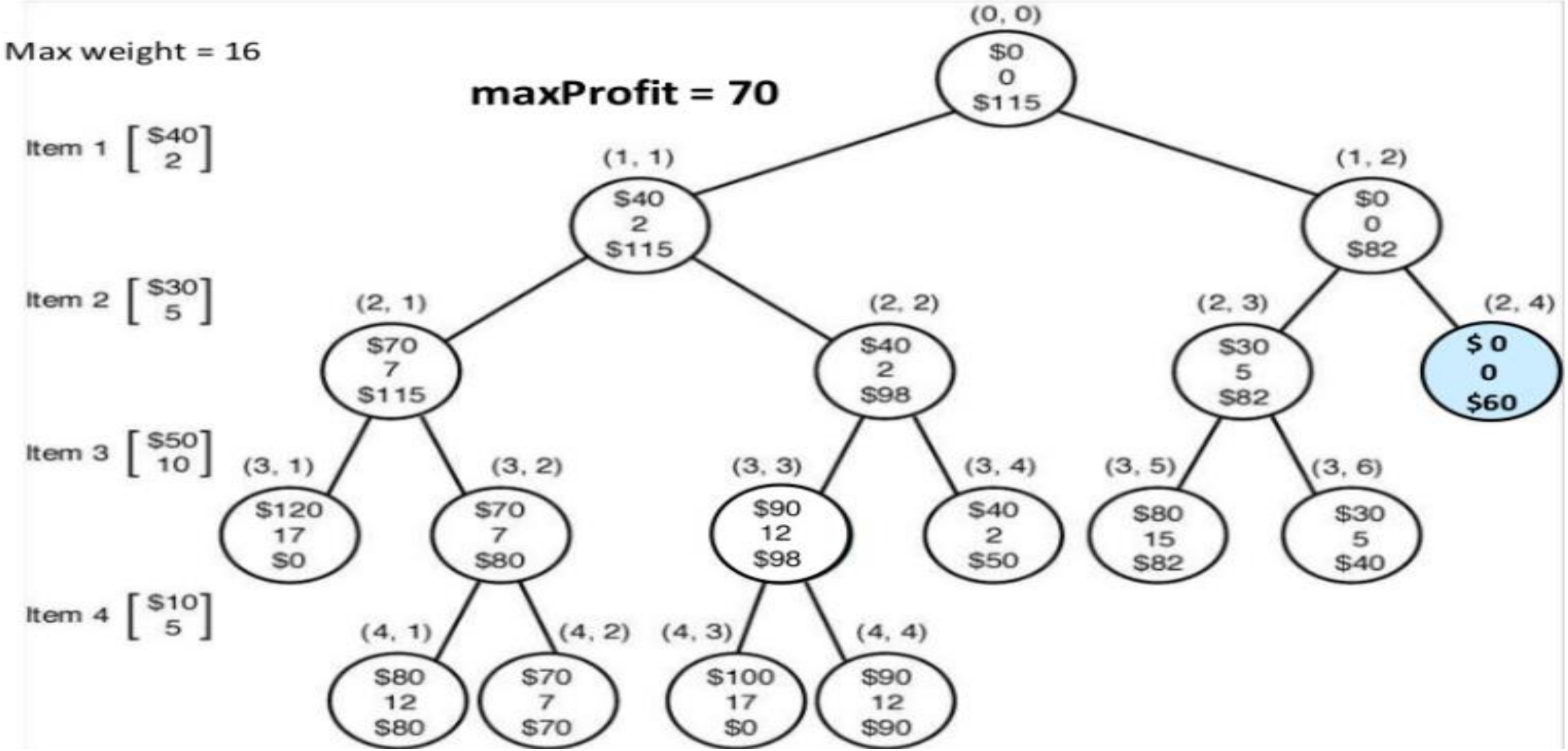

Max weight = 16
maxProfit = 40
Item 1 [$40 2]
Item 2 [$30 5]
Item 3 [$50 10]
Item 4 [$10 5]
(0, 0) $0 0 $115
(1, 1) $40 2 $115
(1, 2) $0 0 $82
(2, 1) $70 7 $115
(2, 2) $40 2 $98
(2, 3) $30 5 $82
(2, 4) $0 0 $60
(3, 1) $120 17 $0
(3, 2) $70 7 $80
(3, 3) $90 12 $98
(3, 4) $40 2 $50
(3, 5) $80 15 $82
(3, 6) $30 5 $40
(4, 1) $80 12 $80
(4, 2) $70 7 $70
(4, 3) $100 17 $0
(4, 4) $90 12 $90


Max weight = 16
maxProfit = 40
Item 1 [$40 2]
Item 2 [$30 5]
Item 3 [$50 10]
Item 4 [$10 5]
(0, 0) $0 0 $115
(1, 1) $40 2 $115
(1, 2) $0 0 $82
(2, 1) $70 7 $115
(2, 2) $40 2 $98
(2, 3) $30 5 $82
(2, 4) $0 0 $60
(3, 1) $120 17 $0
(3, 2) $70 7 $80
(3, 3) $90 12 $98
(3, 4) $40 2 $50
(3, 5) $80 15 $82
(3, 6) $30 5 $40
(4, 1) $80 12 $80
(4, 2) $70 7 $70
(4, 3) $100 17 $0
(4, 4) $90 12 $90


Max weight = 16
maxProfit = 70
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
(0, 0) $0 0 $115
(1, 1) $40 2 $115
(1, 2) $0 0 $82
(2, 1) $70 7 $115
(2, 2) $40 2 $98
(2, 3) $30 5 $82
(2, 4) $0 0 $60
(3, 1) $120 17 $0
(3, 2) $70 7 $80
(3, 3) $90 12 $98
(3, 4) $40 2 $50
(3, 5) $80 15 $82
(3, 6) $30 5 $40
(4, 1) $80 12 $80
(4, 2) $70 7 $70
(4, 3) $100 17 $0
(4, 4) $90 12 $90


Max weight = 16
maxProfit = 70
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
(0, 0)
$0
0
$115
(1, 1)
$40
2
$115
(1, 2)
$0
0
$82
(2, 1)
$70
7
$115
(2, 2)
$40
2
$98
(2, 3)
$30
5
$82
(2, 4)
$0
0
$60
(3, 1)
$120
17
$0
(3, 2)
$70
7
$80
(3, 3)
$90
12
$98
(3, 4)
$40
2
$50
(3, 5)
$80
15
$82
(3, 6)
$30
5
$40
(4, 1)
$80
12
$80
(4, 2)
$70
7
$70
(4, 3)
$100
17
$0
(4, 4)
$90
12
$90


Max weight = 16
maxProfit = 70
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
(0, 0) $0 0 $115
(1, 1) $40 2 $115
(1, 2) $0 0 $82
(2, 1) $70 7 $115
(2, 2) $40 2 $98
(2, 3) $30 5 $82
(2, 4) $0 0 $60
(3, 1) $120 17 $0
(3, 2) $70 7 $80
(3, 3) $90 12 $98
(3, 4) $40 2 $50
(3, 5) $80 15 $82
(3, 6) $30 5 $40
(4, 1) $80 12 $80
(4, 2) $70 7 $70
(4, 3) $100 17 $0
(4, 4) $90 12 $90


Max weight = 16
maxProfit = 70
Item 1 [$40, 2]
Item 2 [$30, 5]
Item 3 [$50, 10]
Item 4 [$10, 5]
(0, 0) $0 0 $115
(1, 1) $40 2 $115
(1, 2) $0 0 $82
(2, 1) $70 7 $115
(2, 2) $40 2 $98
(2, 3) $30 5 $82
(2, 4) $ 0 0 $60
(3, 1) $120 17 $0
(3, 2) $70 7 $80
(3, 3) $90 12 $98
(3, 4) $40 2 $50
(3, 5) $80 15 $82
(3, 6) $30 5 $40
(4, 1) $80 12 $80
(4, 2) $70 7 $70
(4, 3) $100 17 $0
(4, 4) $90 12 $90

Data items used in the Algorithm :

```
struct Item
{
        float weight;
        int value;
}
```

Node structure to store information of decision tree

```
struct Node
{
        int level, profit, bound;
        float weight;
   // level    ---> Level of node in decision tree (or index ) in arr[]
   // profit   ---> Profit of nodes on path from root to this node (including this node)
   // bound ---> Upper bound of maximum profit in subtree of this node
}
```

Algorithm for maxProfit :

```
knapsack(int W, Item arr[], int n)
    queue<Node> Q
    Node u, v                                   //u.level = -1
    Q.push(u)                                   //u.profit = u.weight = 0
    while ( !Q.empty() )                        //int maxProfit = 0
        u = Q.front()  &  Q.pop()
        v.level = u.level + 1                   // selecting the item
        v.weight = u.weight + arr[v.level].weight
        v.profit = u.profit + arr[v.level].value
        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit
        v.bound = bound(v, n, W, arr)
        if (v.bound > maxProfit)
            Q.push(v)
        v.weight = u.weight                     // not selecting the item
        v.profit = u.profit
        v.bound = bound(v, n, W, arr)
        If (v.bound > maxProfit)
            Q.push(v)
    return (maxProfit)
```

Procedure to calculate upper bound :

```
bound(Node u, int n, int W, Item a[])
    if (u.weight >= W)
        return (0)
    int u_bound <- u.profit
    int j <- u.level + 1
    int totweight <- u.weight
    while ((j < n) && (totweight + a[j].weight <= W))
        totweight <- totweight + a[j].weight
        u_bound <- u_bound + a[j].value
        j++
    if (j < n)
        u_bound <- u_bound + ( W - totweight ) * a[j].value /a[j].weight
    return (u_bound)
```

Why branch and bound ?

- Greedy approach works only for fractional knapsack problem.
- If weights are not integers , dynamic programming will not work.
- There are 2n possible combinations of item , complexity for brute force goes exponentially.

A Greedy approach

This is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.

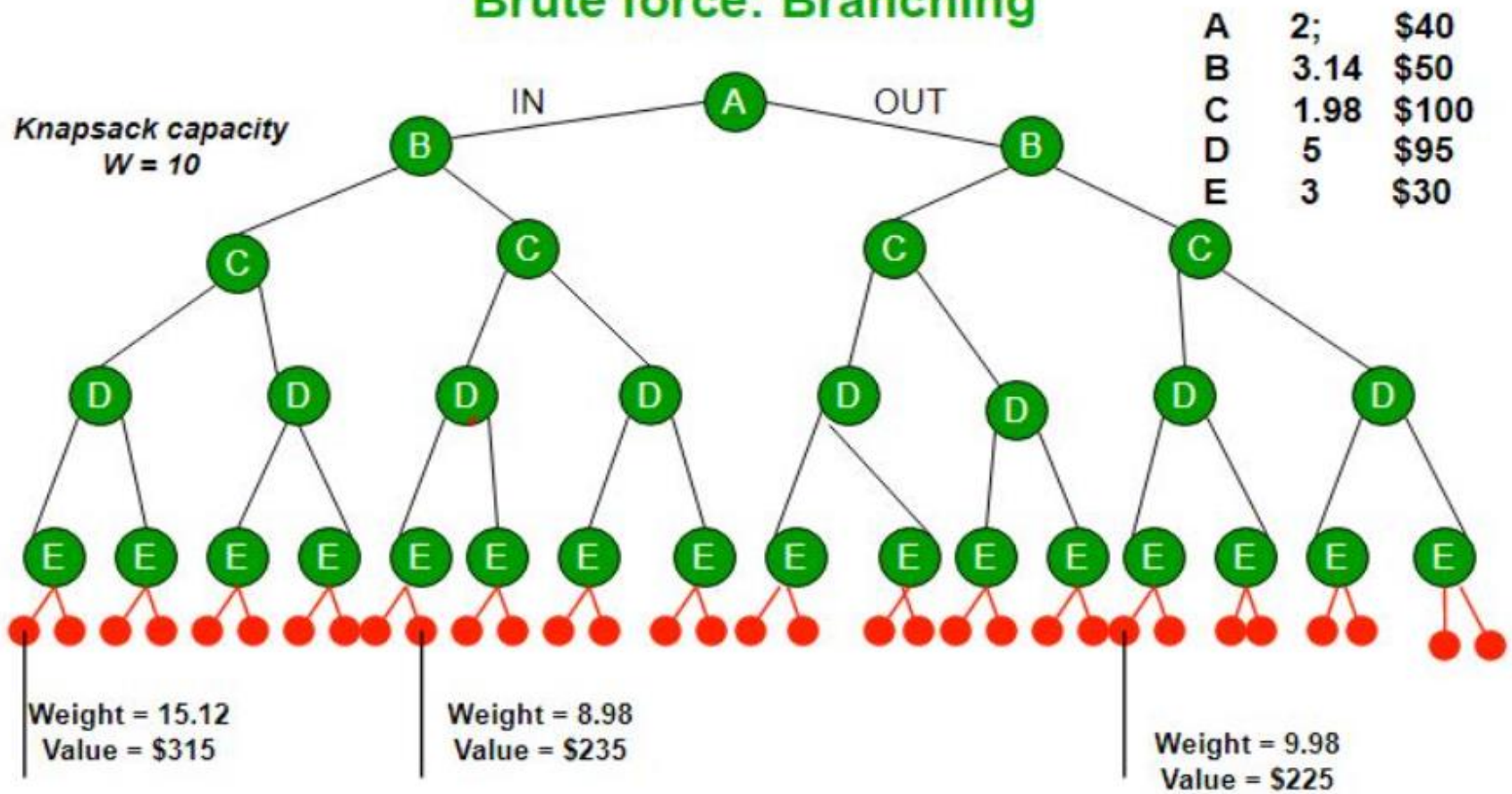
Dynamic Programming (DP) for 0/1 Knapsack problem.

In DP, use a 2D table of size n x W. The DP Solution doesn't work if item weights are not integers.

Brute Force

With n items, there are 2^n solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that satisfies constraint. This solution can be expressed as **tree**.

Brute force: Branching



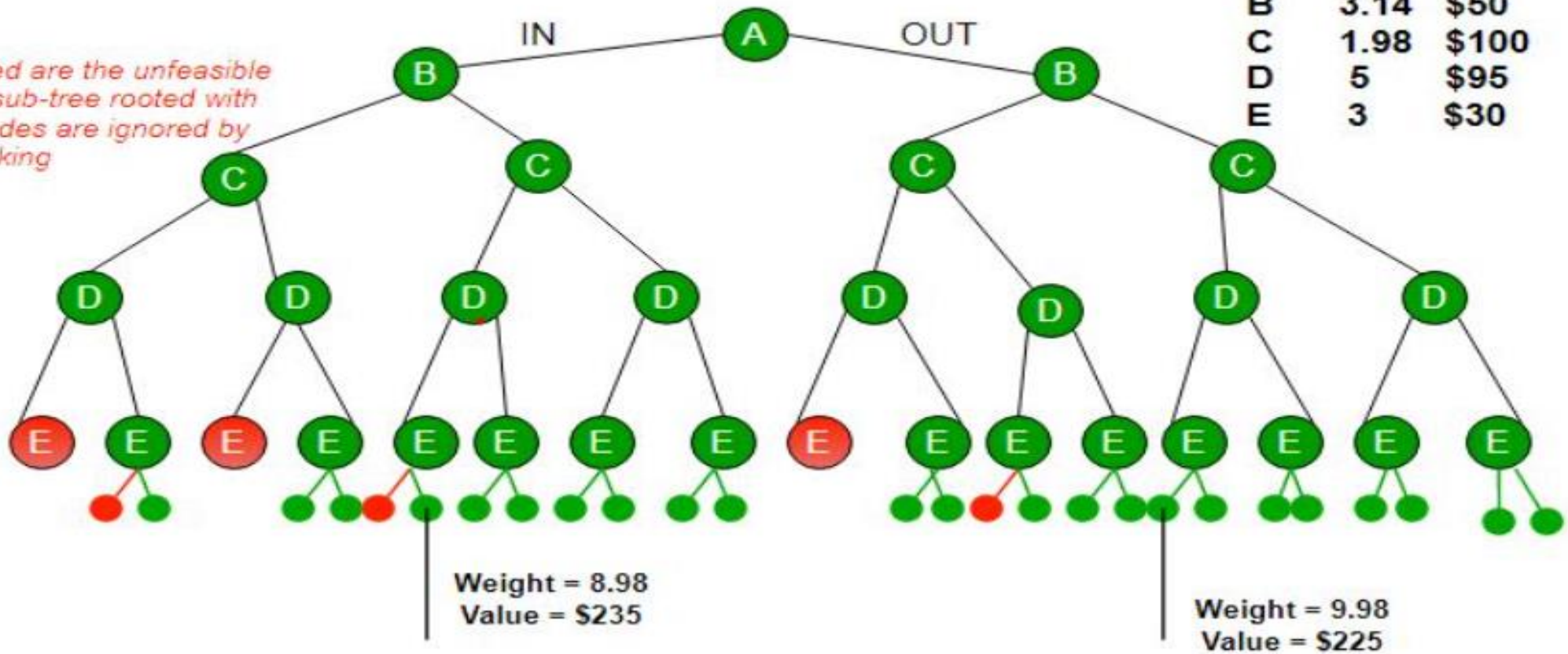
Backtracking is used to optimize the Brute Force solution. In the tree representation, do DFS of tree. If a point is reached where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if more items or a smaller knapsack capacity is used.

Backtracking

Knapsack capacity
W = 10

A	2;	$40
B	3.14	$50
C	1.98	$100
D	5	$95
E	3	$30

Red noted are the unfeasible nodes , sub-tree rooted with these nodes are ignored by backtracking

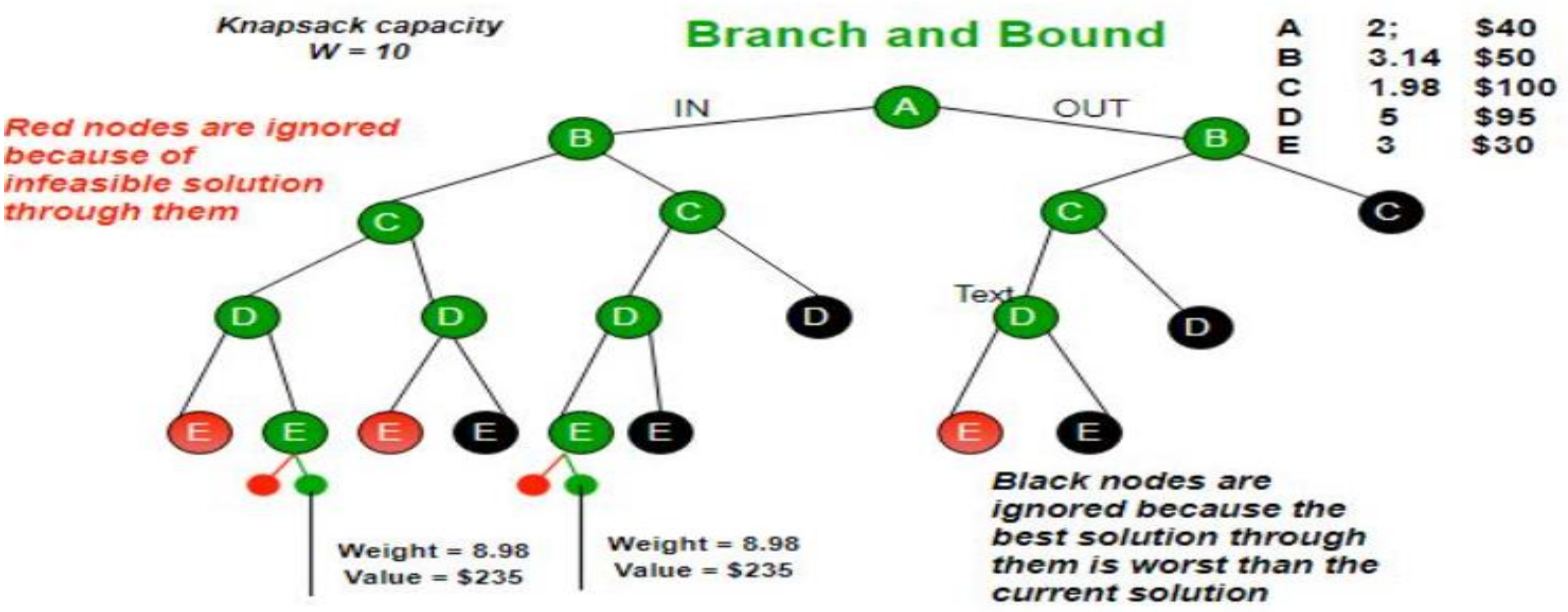


Branch and Bound

The backtracking based solution works better than brute force by ignoring infeasible solutions. It will be better than backtracking if there is a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, ignore this node and its subtrees. Now compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example

Bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.



Hamiltonian Circuits

What is Hamiltonian Circuit?

- Hamiltonian circuit is also known as **Hamiltonian Cycle**.

- If there exists a walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges and returns to the starting vertex, then such a walk is called as a Hamiltonian circuit.

OR

- If there exists a **Cycle** in the connected graph that contains all the vertices of the graph, then that cycle is called as a Hamiltonian circuit.

OR

- A Hamiltonian path which starts and ends at the same vertex is called as a Hamiltonian circuit.

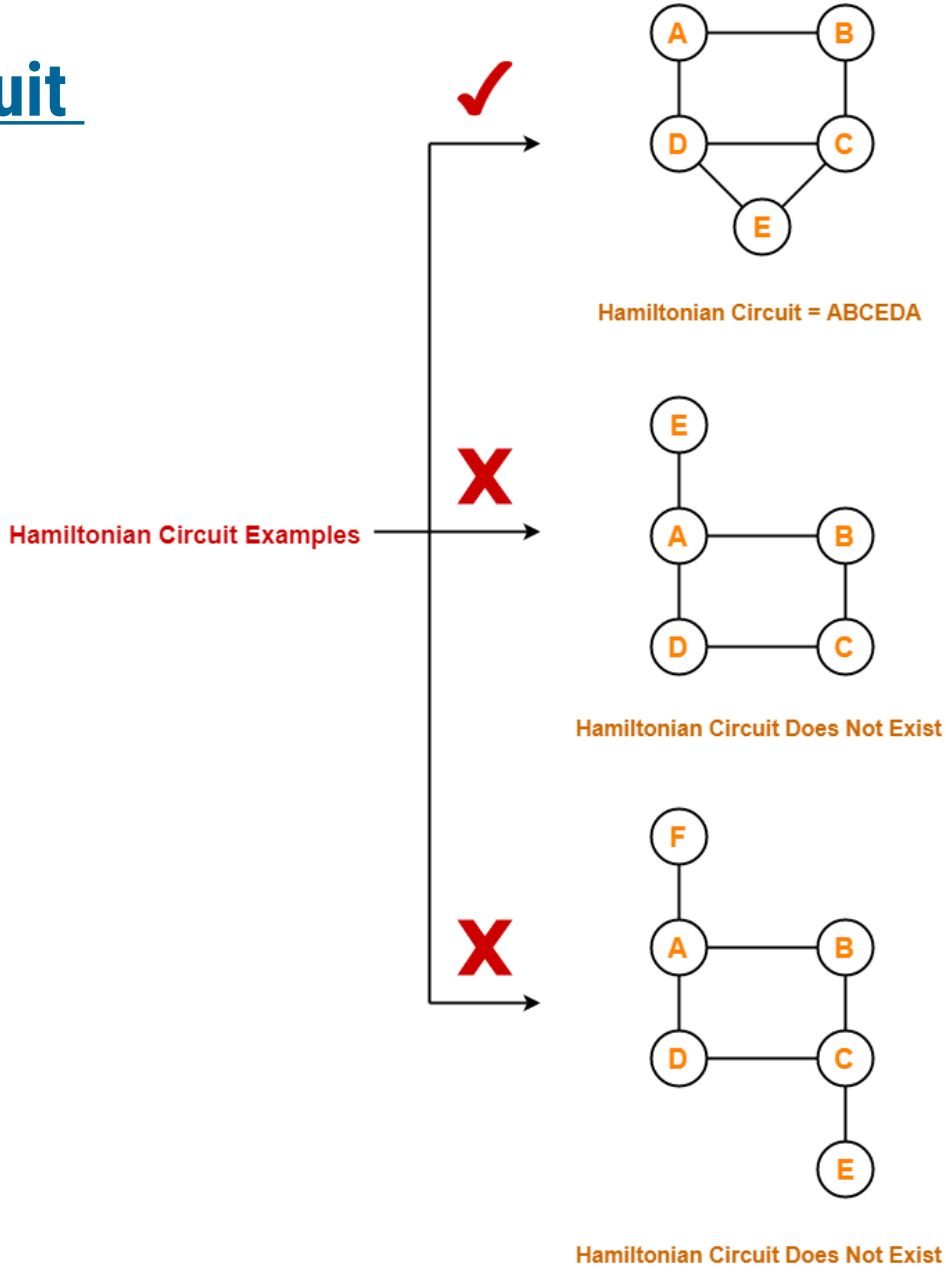
OR

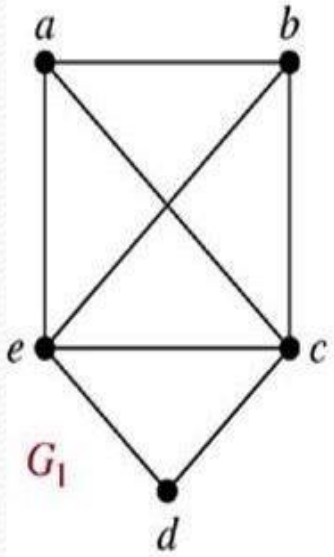
- A closed Hamiltonian path is called as a Hamiltonian circuit.

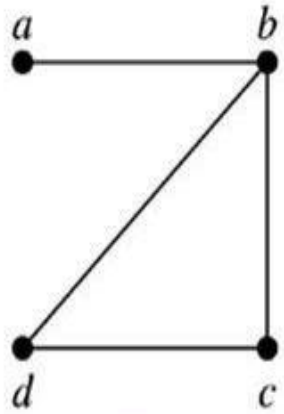
Important:

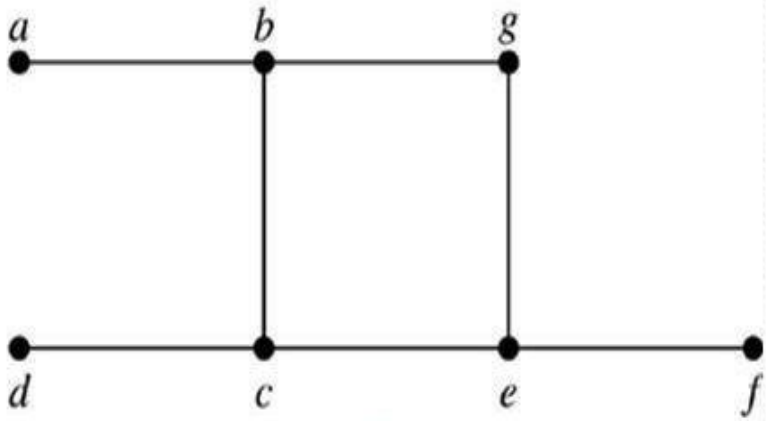
- Any Hamiltonian circuit can be converted to a Hamiltonian path by removing one of its edges.
- Every graph that contains a Hamiltonian circuit also contains a Hamiltonian path but vice versa is not true.
- There may exist more than one Hamiltonian paths and Hamiltonian circuits in a graph.

Hamiltonian Circuit Example :





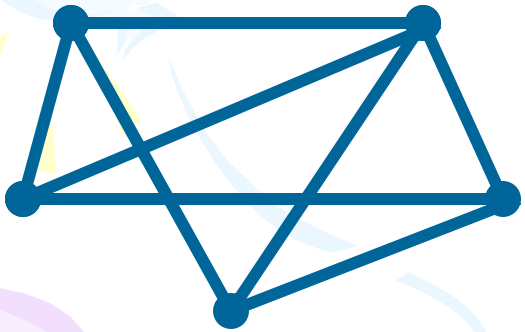


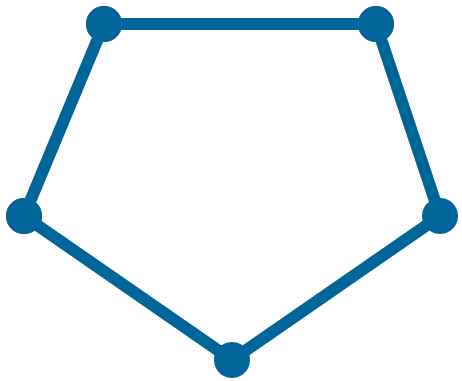


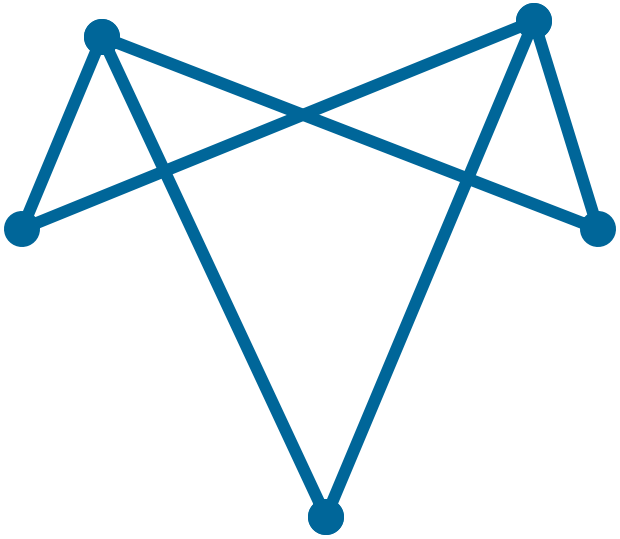
- G_1 has a Hamilton circuit: a, b, c, d, e, a
- G_2 does not have a Hamilton circuit, but does have a Hamilton path: a, b, c, d
- G_3 has neither.

You Try

- Try to find the Hamiltonian circuit in each of the graphs below.







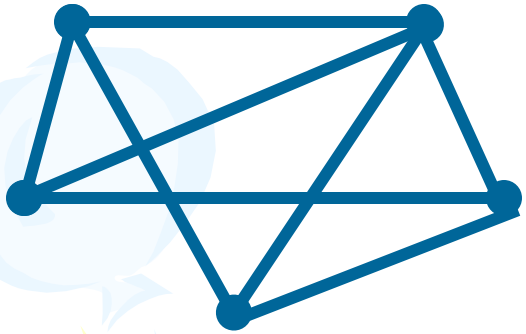
Hamiltonian Theorem

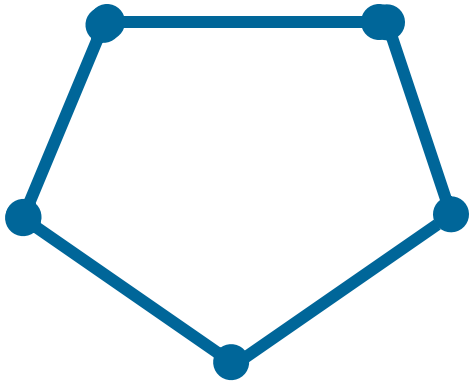
- This theorem guarantees the existence of a Hamilton circuit for certain kinds of graphs.

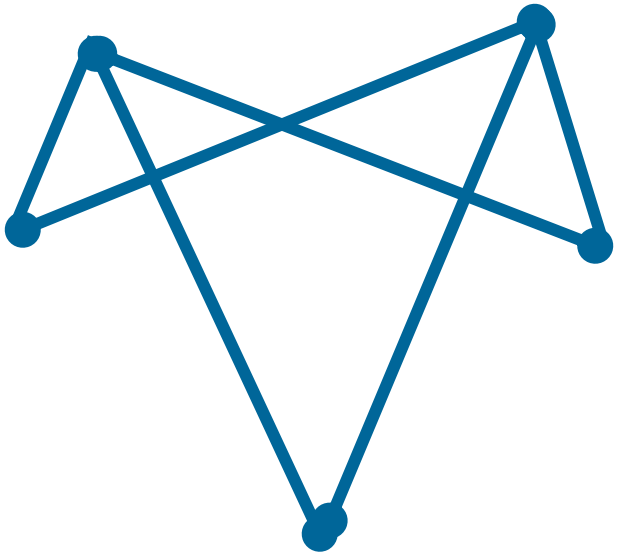
- If a connected graph has n vertices, where n>2 and each vertex has degree of at least n/2, then the graph has a Hamilton circuit.

Degrees

- Check the degrees of the figures in the graphs below.







Since each of the five vertices of the graph has degrees of at least 5/2, the graph has a Hamiltonian circuit.

Hamiltonian Circuits

- If a graph has some vertices with degree less than n/2, the theorem does not apply.
- The second two of the figures that are drawn have vertices that have a degree less than 5/2, so no conclusion can be drawn.
- By inspection, the second figure has a Hamiltonian circuit but the last figure does not.

Comparison to Euler Circuits

- As with Euler circuits, it often is useful for the edges of the graph to have a direction.
- If we consider a competition where every player must play every other player.
- This can be shown by drawing a complete graph where the vertices represent the players.

Competition Example

- In this situation, a directed arrow from vertex A to vertex B would mean that player A defeated player B.
- This type of digraph is known as a **tournament**.
- One interesting property of such a digraph is that every tournament contains a Hamilton path which implies that at the end of the tournament it is possible to rank the teams in order, from winner to loser.

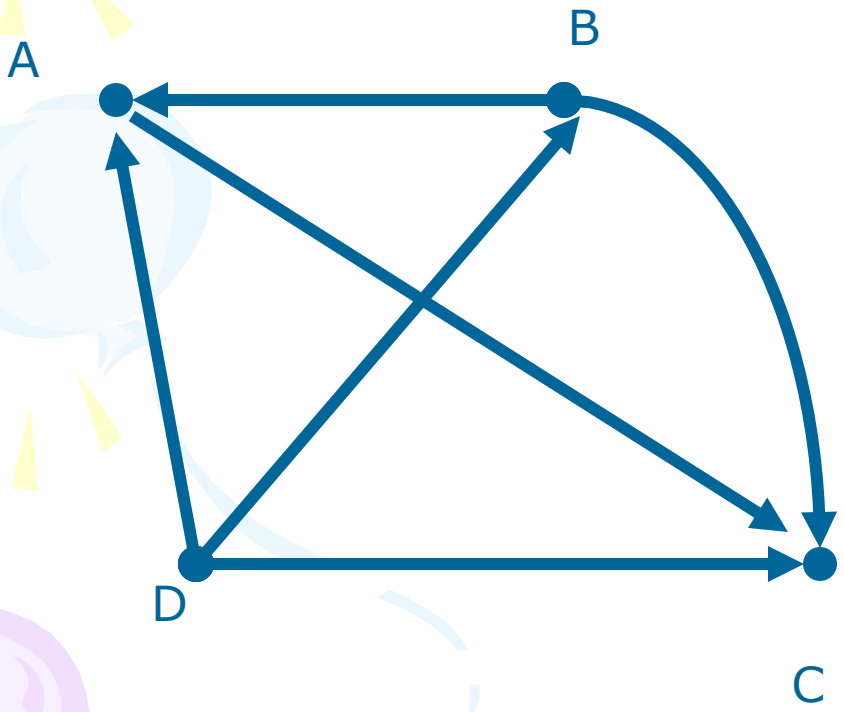
Example

- Suppose Four teams play in the school soccer round robin tournament. The results are as follows:

Game	AB	AC	AD	BC	BD	CD
Winner	B	A	D	B	D	D

- Draw a digraph to represent the tournament. Find a Hamiltonian path and then rank the participants from winner to loser.

Example (cont'd)



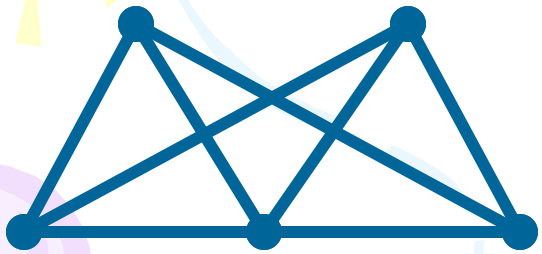
- Remember that a tournament results from a complete graph when direction is given to the edges.
- There is only one Hamiltonian path for this graph, DBAC.
- Therefore, D is first, B is second, A is third and C is fourth.

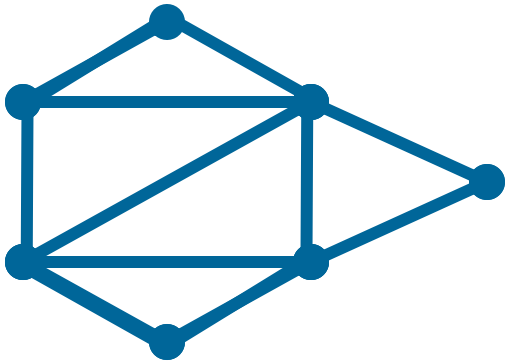
Practice Problems

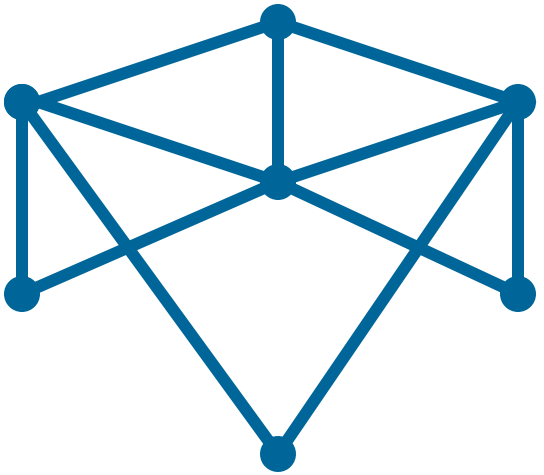
1. a. Construct a graph that has both an Euler and a Hamiltonian circuit.

   b. Construct a graph that has neither an Euler now a Hamiltonian circuit.

2. Apply the Hamiltonian theorem to the graphs below and indicate which have Hamiltonian circuits. Explain why.







Practice Problems (cont'd)

3. Hamilton's Icosian game was played on a wooden regular dodecahedron. In the planar representation of the game, find a Hamiltonian circuit for the graph. Is there only one Hamiltonian circuit for the graph? Can the circuit begin at any vertex?

Practice Problems (cont'd)

4. Draw a tournament with 3 vertices in which
   a. One player wins all of the games it plays.
   b. Each player wins one game.
   c. Two players lose all of the games they play.

5. Draw a tournament with five players, in which player A beats everyone, B beats everyone but A, C is beaten by everyone and D beats E.

6.Find all the directed Hamiltonian paths for the following tournaments: