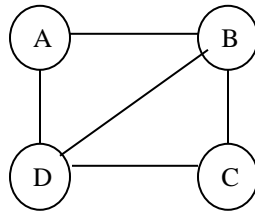


Graph

A graph is a collection of nodes called vertices, and the connections between them, called edges. Formally, a graph is a set of vertices and a binary relation between vertices, adjacency. The following figure represents a graph.



Vertices, $v = \{A, B, C, D\}$

Edge, $E = \{AB, AD, BD, BC, CD\}$

Graph $\equiv (V, E)$

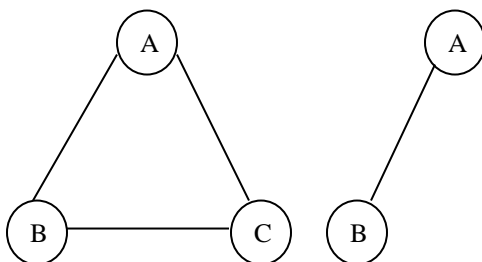
Application of graph:

- Network of communication
- Data organization
- Computational devices
- The flow of computation

Different Types of Graph

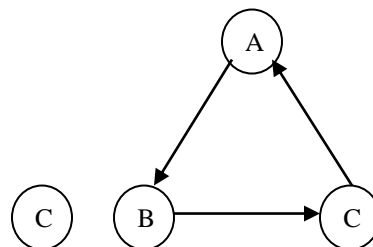
There are several types of graph. They are as follows:

1. Connected graph
2. Disconnected graph
3. Directed graph
4. Undirected
5. Weighted graph
6. Unweighted graph
7. Cyclic graph
8. Acyclic graph
9. Bipartite graph
10. Complete Bipartite graph



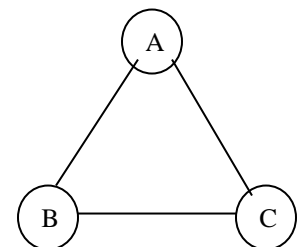
Connected graph

Disconnected graph



Edges, $E = \{AB, BC, CA\}$

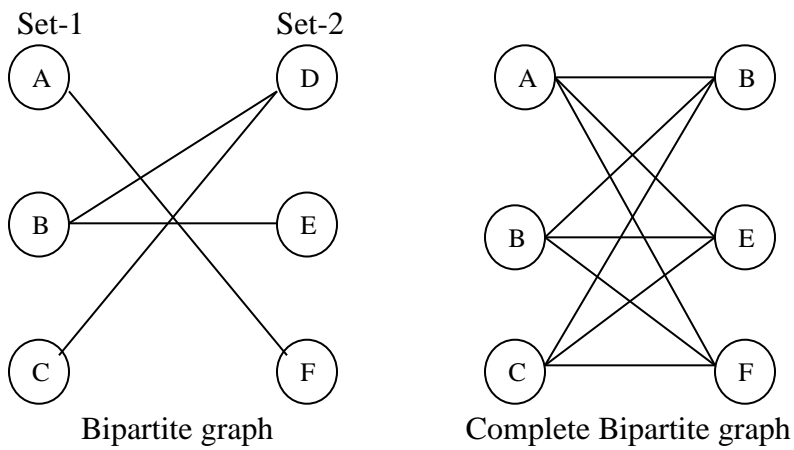
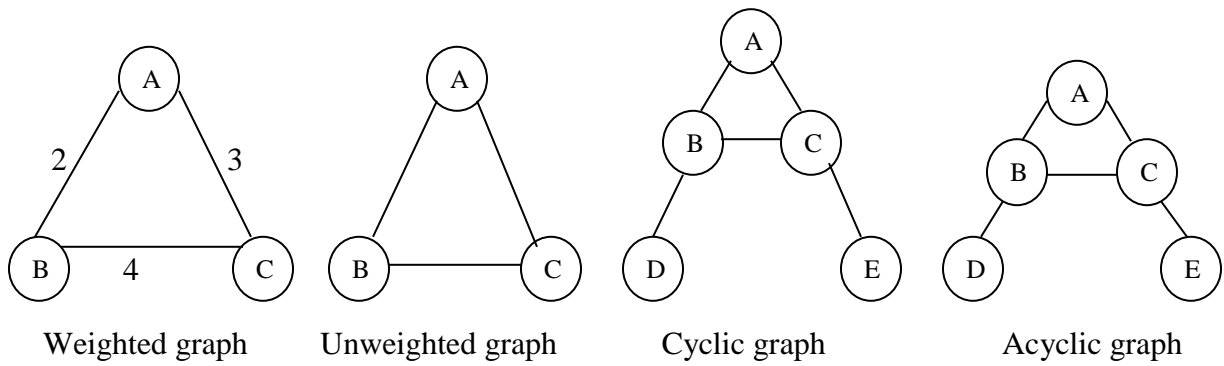
Directed graph



Edges, $E = \{AB, BC, CA\}$

or Edges, $E = \{AB, BC, CA\}$

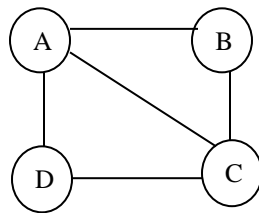
Undirected graph



Graph Adjacency Structure

The two types graph representation technique is in the following.

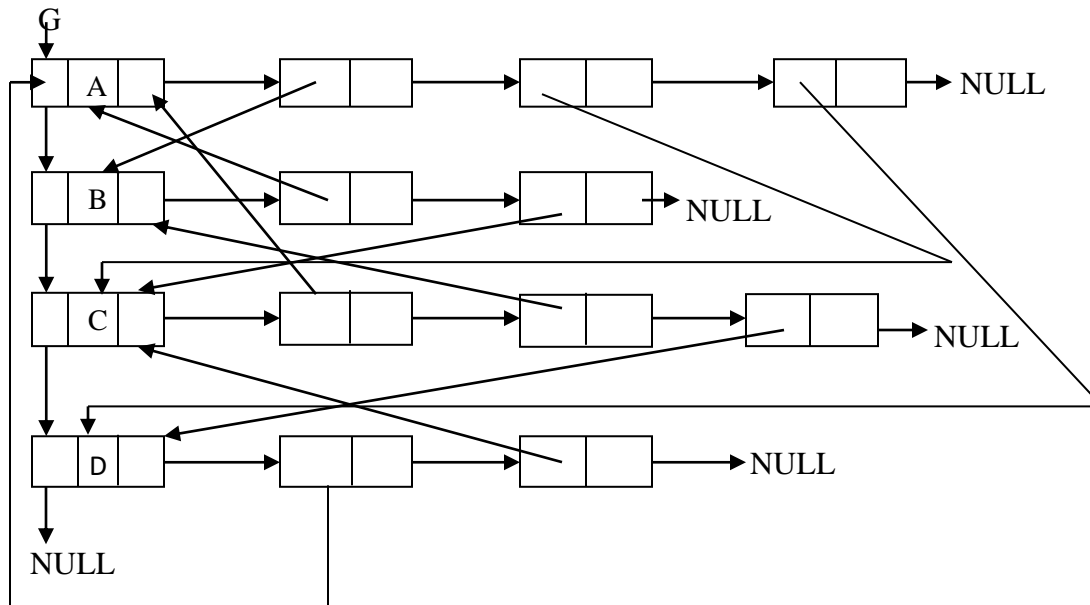
- Array (2D)
- Linked List



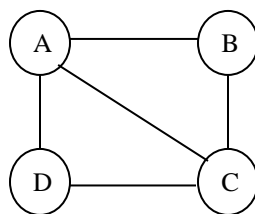
The graph representation according to array:

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	1
D	1	0	1	0

The graph representation according to linked list:



Relationship between degree and edges:



Degree of vertex A=3

Degree of vertex B=2

Degree of vertex C=3

Degree of vertex D=2

Total degree = 10 = 2 * 5

= 2 * total number of edges

Therefore, total degree = 2 * total number of edges

Graph Traversal Algorithm

Commonly, we will discuss some graph algorithms such as BFS, DFS, topological ordering, single source shortest path and nearest neighbor algorithms in this chapter. A breadth-first search is when you inspect every node on a level starting at the top of the tree and then move to the next level. A depth-first search is where you search deep into a branch and do not move to the next one until you have reached the end. A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering.

Breadth First Search (BFS)

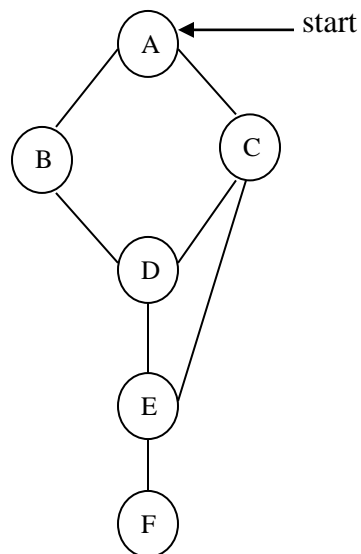
Algorithm of BFS:

```

Queue:=start
while(queue!=empty)
    v:=queue
    if (v is not visited)
        visit v
        visit sequence:=v
        for(all non-visited adjacent u of v)
            queue:=u
        end for
    end if
end while

```

Mechanism:



queue

A			
--------------	--	--	--

v:=queue visit v visited sequence:=v u:=non-visited adjacent u of v

v='A' A A u=B, C

v='B' B AB u=~~A~~, D

v='C' C ABC u=~~A~~, D, E

v='D' D ABCD u=~~B~~, ~~C~~, E

queue :=u

B	C		
--------------	---	--	--

C	D		
--------------	---	--	--

D	D	E	
--------------	---	---	--

D	E	E	
--------------	---	---	--

v='D' × ABCD

×

v='E' E ABCDE

u=~~D~~, F

v='E' × ABCDE

×

v='F' F ABCDEF

u=~~E~~

E	E		
--------------	---	--	--

E	F		
--------------	---	--	--

F			
--------------	--	--	--

--	--	--	--

The time complexity of Breadth-First Search (BFS) algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. Every vertex (V) and every edge (E) is explored only once in BFS technique.

Write a program for BFS algorithm

```
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
char que[20];
int front=0, rear=0, n;
char arr[20];
int bfs(int );
char ajMat[20][20];
char b[20];
void display();
int p=0;
int main( ){
    char v;
    int i,j;
    printf("Enter the number of nodes in a graph:=");
    scanf("%d",&n);

    for( i=0; i<n; i++) {
        printf("\nEnter the value of node of graph:%d=",i);
        scanf("%s",&b[i]);
    }
    printf("Enter the value in adjacency matrix in from of 'y' or 'n':=\n");
    printf("If there exists an edge between two vertices than 'y' otherwise 'n':=\n");
    for( i=0; i<n; i++)
        printf(" %c",b[i]);
    for( i=0;i<n; i++)
    {
        printf("\n%c ",b[i]);
        for( j=0; j<n; j++)
        {
```

```

        printf("%c ",v=getchar());
        ajMat[i][j]=v;
    }
    printf("\n\n");
}
for( i=0;i<n;i++)
bfs(i);

display();
getchar();
return 0;
}
void display( ){
    printf("BFS of Graph : ");
    for(int i=0; i<n; i++)
        printf("%c ",arr[i]);
}
void insert(char val)
{
    que[front]=val;
    front++;
}
char del( ){
    rear=rear+1;
    return que[rear-1];
}
bool unVisit(char val)
{
    for(int i=0; i<front; i++)
    {
        if(val==que[i])
            return false;
    }
}
return true;
}
int bfs(int i){
    char m;
    if(front==0)    {
        insert(b[i]);
    }
    for(int j=0; j<n; j++)
    {
        if(ajMat[i][j]=='y')    {
            if(unVisit(b[j]))

```

```

        {
            insert(b[j]);
        }    }    }
m=del();
arr[p]=m;
p++;
return 0;
}

```

Depth First Search (DFS)

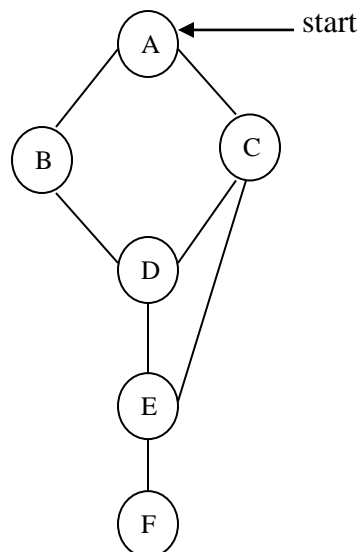
Algorithm of DFS:

```

Stack:=start
while(stack!=empty)
    v:=stack
    if (v is not visited)
        visit v
        visit sequence:=v
        for(all non-visited adjacent u of v)
            stack:=u
        end for
    end if
end while

```

Mechanism



queue

A			
--------------	--	--	--

<u>v<=stack</u>	<u>visit v</u>	<u>visited sequence</u>	<u>u=non visited-adjacent vertices u of v</u>	<u>stack<=u</u>				
v='A'	A	A	u=B, C	<table><tr><td>B</td><td>C</td><td></td><td></td></tr></table>	B	C		
B	C							
v='C'	C	AC	u= A , D, E	<table><tr><td>B</td><td>D</td><td>E</td><td></td></tr></table>	B	D	E	
B	D	E						
v='E'	E	ACE	u= C , D, F	<table><tr><td>B</td><td>D</td><td>D</td><td>F</td></tr></table>	B	D	D	F
B	D	D	F					
v='F'	F	ACEF	u= E	<table><tr><td>B</td><td>D</td><td>D</td><td></td></tr></table>	B	D	D	
B	D	D						
v='D'	D	ACEF D	u= B, C , E	<table><tr><td>B</td><td>D</td><td>B</td><td></td></tr></table>	B	D	B	
B	D	B						
v='B'	B	ACEF DB	u= A , D	<table><tr><td>B</td><td>B</td><td></td><td></td></tr></table>	B	B		
B	B							
v='D'	×	ACEF DB	×	<table><tr><td>B</td><td></td><td></td><td></td></tr></table>	B			
B								
v='B'	×	ACEFDB	×	<table><tr><td></td><td></td><td></td><td></td></tr></table>				

The time complexity of Depth-First Search (DFS) algorithm is **O (V + E)**, where V is the number of vertices and E is the number of edges in the graph. Every vertex (V) and every edge (E) is explored only once in DFS technique.

Write a program in C for DFS algorithm

```
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
char stack[20];
int top=-1, n;
char arr[20];
char dfs(int );
char ajMat[20][20];
char b[20];
void display();
int p=0,i;
int main()
{
    char v;
    int l=0;
    printf("Enter the number of nodes in a graph:=");
```



```

scanf("%d",&n);

for(i=0; i<n; i++)
{
    printf("\nEnter the value of node of graph:%d=",i);
    scanf("%s",&b[i]);
}
char k=b[0];
printf("Enter the value in adjacency matrix in from of 'Y' or 'N':\n");
printf("\nIf there is an edge between the two vertices then enter 'Y' or 'N':=\n");
for(i=0; i<n; i++)
printf(" %c ",b[i]);
for(i=0;i<n; i++)
{
    printf("\n%c ",b[i]);
    for(int j=0; j<n; j++)
    {
        printf("%c ",v=getchar());
        ajMat[i][j]=v;
    }
    printf("\n\n");
}
for(i=0;i<n;i++)
{
    l=0;
    while(k!=b[l])
        l++;
    k=dfs(l);
}
display();
getchar();
return 0;
}
void display()
{
    printf(" DFS of Graph : ");
    for(int i=0; i<n; i++)
        printf("%c ",arr[i]);
}
void push(char val)
{
    top=top+1;
    stack[top]=val;
}
char pop()
{
    return stack[top];
}

```

```

bool unVisit(char val)
{
    for(i=0; i<p; i++)
        if(val==arr[i])
            return false;
    for(i=0; i<=top; i++)
        if(val==stack[top])
            return false;
    return true;
}
char dfs(int i)
{
    char m;
    if(top==-1)
    {
        push(b[i]);
    }
    m=pop();
    top--;
    arr[p]=m;
    p++;
    for(int j=0; j<n; j++)
    {
        if(adjMat[i][j]=='y')
        {
            if(unVisit(b[j]))
            {
                push(b[j]);
            }
        }
    }
    return stack[top];
}

```

Output:

Enter the number of nodes in a graph:=6

Enter the value of node of graph:=a b c d e f

Enter the value in adjacency matrix in from of 'Y' or 'N':

If there is an edge between the two vertices then enter 'Y' or 'N':=

	a	b	c	d	e	f
a	n	y	y	n	n	n
b	y	n	n	y	n	n
c	y	n	n	y	y	n
d	n	y	y	n	y	n
e	n	n	y	y	n	y
f	n	n	n	n	y	n

DFS of graph: a c e f d b

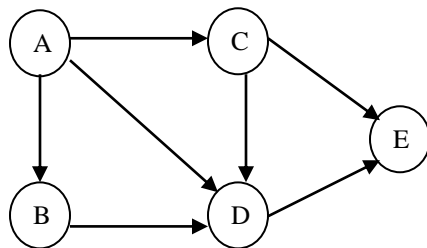
Topological Sorting

A topological sort (sometimes abbreviated topsort or toposort) or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge $\{u, v\}$ from vertex u to vertex v , u comes before v in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG).

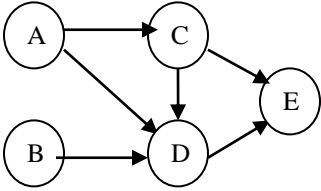
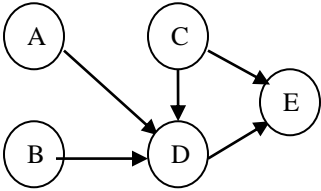
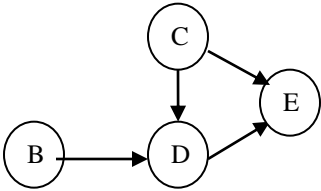
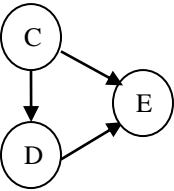
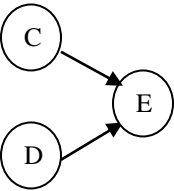
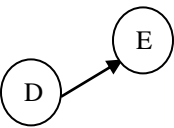
Algorithm:

```
L<-Topological order// initially L is empty
S<-set of all nodes with no incoming edges
while( S !=empty)
    remove a node 'm' from S
    insert m into L
    for(each node n with an edge e from m to n)
        remove edge e from the graph
        if (n has no other incoming edge)
            insert n into S
        end if
    end for
end while
return L
```

The mechanism of topological ordering is in the following:



```
order, L={ }
set, S={A}
m=source vertex
n=target vertex
```

<u>m<=set</u>	<u>order(L<=m)</u>	<u>edge (e)</u>	<u>disconnect mn from graph</u>	<u>set <=n</u>
m='A'	order={ A }	AB		set={ B }
		AC		set={ B, C }
		AD		set= { B , C }
m='B'	order={ A, B }	BD		set={ C }
m='C'	order={ A, B, C }	CD		set={ D }
		CE		set={ D }
m='D'	order={ A, B, C, D }	DE	No graph	set={ E }
m='E'	order={ A, B, C, D, E }	×	No graph	set={ }

Therefore, topological order:=A->B->C->D->E

Note: The vertex where no edge is directing to is kept in set.

Single Source Shortest Path

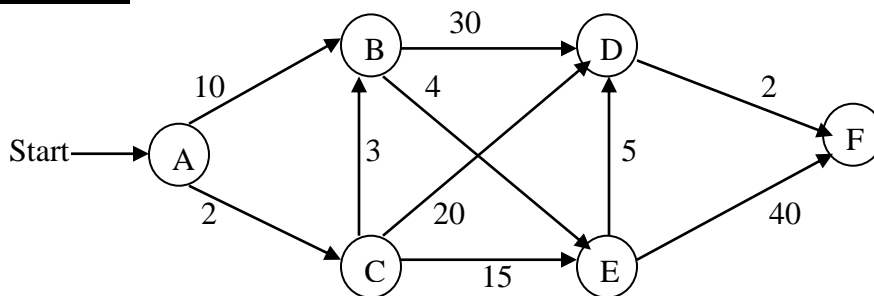
Time Complexity of Single Source Shortest Path: Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O((V + E) \log V)$.

Single Source Shortest Path: Dijkstra's Algorithm

Algorithm:

```
Procedure Dijkstra(cost, vertex, n, source)
set={source}
for{all v in vertex}
dist[v]=cost[source][v]
end for
for i=1 to n-1
    select a vertex u such that u is not in set and dist(u) is minimum
    set={u}
    for (all v that are not in set)
        dist[v]=min{dist[v], dist[u]+cost[u][v]}
    end for
end for
end procedure
```

Mechanism:



All shortest paths from source node A to all the remaining nodes using the dijkstra's algorithm are in the following:

Step1: Cost Matrix Construction

A	B	C	D	E	F
0	10	2	∞	∞	∞
∞	0	∞	30	4	∞

A	∞	3	0	20	15	∞
B	∞	∞	∞	0	∞	2
C	∞	∞	∞	5	0	40
D	∞	∞	∞	∞	∞	∞
E						
F						

Step2: Minimum Cost Calculation:			[dist[v]=min{dist[v], dist[u]+cost[u][v]}]					
<u>Iteration</u>	<u>selected vertex</u>	<u>set</u>	<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>	<u>E</u>	<u>F</u>
initial	---	{A}	0	10	2	∞	∞	∞
1	C	{AC}	0	5	2	22	17	∞
2	B	{ACB}	0	5	2	22	9	∞
3	E	{ACBE}	0	5	2	14	9	49
4	D	{ACBED}	0	5	2	14	9	16
5	F	{ACBEDF}	0	5	2	14	9	16

Step3: shortest path calculation

<u>Source</u>	<u>Destination</u>	<u>Shortest Path</u>	<u>Cost</u>	<u>Verification</u>
A	B	ACB	5	2+3
A	C	AC	2	2
A	D	ACBED	14	2+3+4+5
A	E	ACBE	9	2+3+4
A	F	ACBEF	16	2+3+4+5+2

Nearest Neighbor Algorithm

The nearest neighbor algorithm was one of the first algorithms used to solve the travelling salesman problem (TSP) approximately. In that problem, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. A nearest neighbor algorithm plots all vectors in a multi-dimensional space and uses each of the points to find a neighboring point that is nearest. Different types of nearest neighbor algorithms consider a neighboring point differently.

Algorithm:

step1: Choose a starting point called vertex v.

step2: Check all the edges connected to v, and choose one that has the smallest weight. Proceed along this edge to the next vertex.

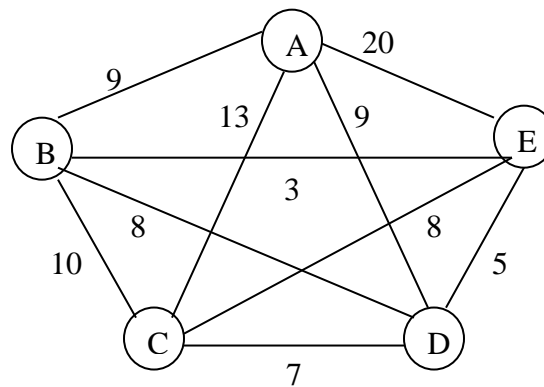
step3: At each vertex can be reached, check the edges from that vertex to vertices who are not yet visited. Choose one that has the smallest weight. Proceed along this edge to the next vertex.

step4: Repeat step 3 until to be visited all the vertices.

step5: Return to the starting vertex.

The working principle of nearest neighbor algorithm is in the following:

Let the following weighted graph with five vertices. We will visit each node approximately the least time using nearest neighbor algorithm.



Case 1:

<u>visit vertex</u>	<u>route</u>	<u>edge</u>	<u>cost</u>		<u>selected edge</u>	<u>selected vertex</u>
A	A->B	AB	9	}	AD	D
	A->E	AE	20			
	A->C	AC	13			
	A->D	AD	9			
D	D->E	DE	5	}	DE	E
	D->B	DB	8			
	D->C	DC	7			
E	E->B	EB	3	}	EB	B
	E->C	EC	8			
B	B->C	BC	10	}	BC	C
C	×	×	×		×	×

Visited sequence \equiv A->D->E->B->C->A

Tour cost = $9+5+3+10+13 = 40$

Case 2:

<u>visit vertex</u>	<u>route</u>	<u>edge</u>	<u>cost</u>		<u>selected edge</u>	<u>selected vertex</u>
A	A->B	AB	9	}	AB	B
	A->E	AE	20			
	A->C	AC	13			
	A->D	AD	9			
B	B->E	BE	3	}	BE	E
	B->D	BD	8			
	B->C	BC	10			
E	E->D	ED	5	}	ED	D
	E->C	EC	8			
D	D->C	DC	7	}	DC	C
C	×	×	×		×	×

Visited sequence \equiv A->B->E->D->C->A

Tour cost = $9+3+5+7+13 = 37$

Note: But the nearest neighbor algorithm does not always provide optimal solution. So, dynamic programming approach is needed to find optimal solution.

Sparse graph and Dense graph

Sparse graph

If a graph has V vertices, the maximum number of edges it can have is $V(V-1)/2$ for an undirected graph and $V(V-1)$ for a directed graph. A graph is considered sparse if it has much fewer edges than this maximum number, typically close to $O(V \log V)$ or $O(V)$ edges.

Dense graph

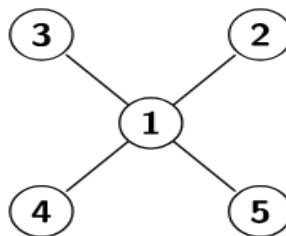
If a graph has V vertices, the maximum number of edges it can have is $V(V-1)/2$ for an undirected graph and $V(V-1)$ for a directed graph. A graph is considered dense if it has close to this maximum number of edges, typically on the order of $O(V^2)$ edges.

Sparse Graph vs. Dense Graph

Graphs can be categorized broadly into sparse and dense based on the number of edges relative to the number of vertices.

- **Sparse Graph:**
 - Has relatively few edges.
 - Typical edge count is $O(V)$ or slightly more like $O(V \log V)$.
 - Example: A network of cities where only a few cities are directly connected by roads.
- **Dense Graph:**
 - Has a large number of edges, close to the maximum possible.
 - Typical edge count is $O(V^2)$.
 - Example: A network of cities where almost all cities are directly connected by roads.

Now that we know how to compute the density of a graph, we can apply it in a practical exercise. We'll take as an example the first graph we encountered in this tutorial:



This graph has a form $G(V, E)$, where $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 3), (1, 4), (1, 5)\}$. Therefore, its first two characteristics are $|V| = 5$ and $|E| = 4$. Because the graph is undirected, we can calculate its maximum number of edges as:

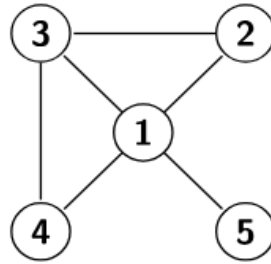
$$\text{Max}_U(V) = \binom{|V|}{2} = \frac{5 \cdot (5-1)}{2} = 10$$

From this, we can then calculate the density of the graph as:

$$D_U(V, E) = \frac{|E|}{\text{Max}_U(V)} = \frac{4}{10} = \frac{2}{5}$$

Because $D_U(V, E) < 1/2$, we can conclude that $G(V, E)$ is a sparse graph.

If, instead, the graph had just two extra edges; say, (2, 3) and (3, 4), then it would look like this:



And the related calculations would change as follows:

$$|E| = 6 \rightarrow D_U(V, E) = \frac{|E|}{\text{Max}_U(V)} = \frac{6}{10} = \frac{3}{5}$$

This, in turn, makes the extended graph a dense graph, because $D_U > 1/2$.

Let us consider number of vertices = v

Number of maximum possible edges (for directed graph) = $v(v-1)/2 = 1/2(v^2 - v)$

Space complexity = $O(v^2)$

For a given graph vertices v and edges e .

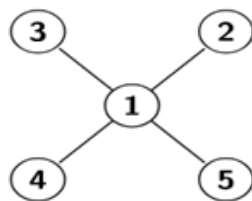
For this graph possible maximum $E_{\max} = v(v-1)/2$

Density of a graph $D = e/E_{\max} = 4/10 = 0.4$

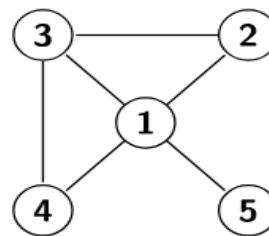
If $D < 1/2$ then the graph is sparse

Density of a graph $D = e/E_{\max} = 6/10 = 0.6$

If $D > 1/2$ then the graph is dense graph



Sparse graph



Dense graph

Graph space complexity

For Dense graph $O(v^2)$

For Sparse graph $O(v)$

Applications of Dense Graphs

Social Networks: In small, tightly-knit communities, where most people know each other.

Communication Networks: Networks where most devices need to communicate directly, such as in mesh networks.

Biological Networks: Gene or protein interaction networks, where many elements are interconnected.

Transportation Systems: In metro systems where many stations are directly connected by routes

Sparse graph

```
#include <bits/stdc++.h>
using namespace std;
// Function to create an adjacency list from an edge list
vector<vector<int>> createAdjList(int n,
                                vector<vector<int>>& edgeList) {
    // Initialize an adjacency list with empty vectors
    // for each vertex
    vector<vector<int>> adjList(n);
    // Populate the adjacency list
    for (const auto& edge : edgeList) {
        int u = edge[0];
        int v = edge[1];
        adjList[u].push_back(v);
        adjList[v].push_back(u); // Assuming an undirected graph
    }
    return adjList;
}

void printGraph(const vector<vector<int>>& adjList) {
    for (int i = 0; i < adjList.size(); ++i) {
        cout << "Vertex " << i << ": ";
        for (const auto& neighbor : adjList[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}

int main() {
    int n = 5;
    vector<vector<int>> edgeList =
        {{0, 1}, {0, 3}, {1, 2}, {3, 4}};
    // Create an adjacency list from the edge list
    vector<vector<int>> adjList = createAdjList(n, edgeList);
```

```

    printGraph(adjList);
    return 0;
}

```

Output:

```

Vertex 0: 1 3
Vertex 1: 0 2
Vertex 2: 1
Vertex 3: 0 4
Vertex 4: 3

```

Dense graph

```

#include <bits/stdc++.h>
using namespace std;
// Function to create an adjacency matrix from an edge list
void createAdjMatrix(int n, vector<vector<int>>& edges,
                    vector<vector<int>>& adjMatrix) {
    // Initialize the adjacency matrix with 0s
    adjMatrix.resize(n, vector<int>(n, 0));
    // Populate the adjacency matrix based on the edges
    for (const auto& edge : edges) {
        int u = edge[0];
        int v = edge[1];
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // For undirected graph
    }
}

void printAdjMatrix(const vector<vector<int>>& adjMatrix) {
    int n = adjMatrix.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int n = 5;
    vector<vector<int>> edges =
        {{0, 1},{0, 2},{1, 2},{1, 3}, {2, 3},{1,4},{3,4},{2, 4}};
    vector<vector<int>> adjMatrix;
    createAdjMatrix(n, edges, adjMatrix);
    printAdjMatrix(adjMatrix);
    return 0;
}

```

```
}
```

Output:

```
0 1 1 0 0
```

```
1 0 1 1 1
```

```
1 1 0 1 1
```

```
0 1 1 0 1
```

```
0 1 1 1 0
```