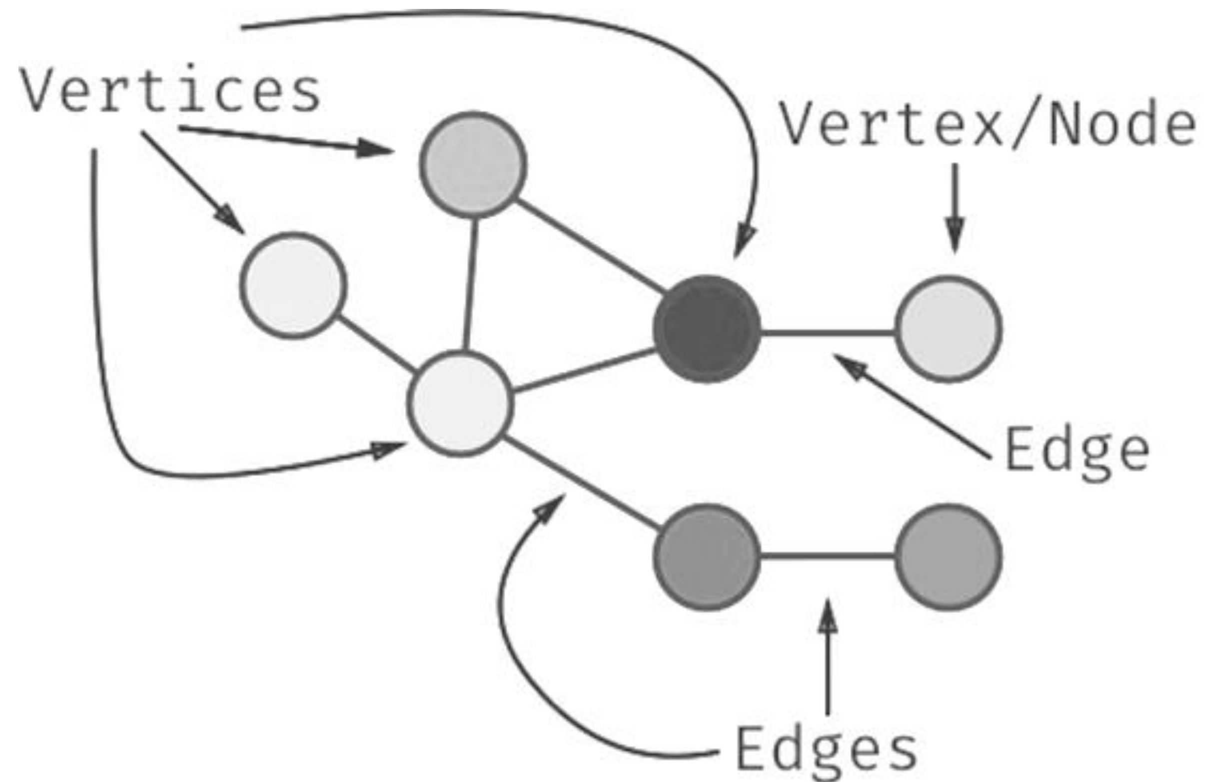
CS 2124: DATA STRUCTURES
Spring 2024



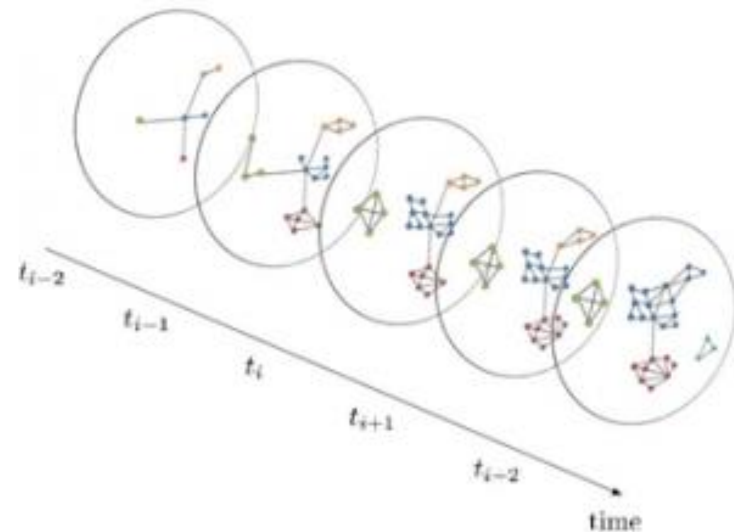
- Lecture 11.2
  - Introduction to Graphs – II

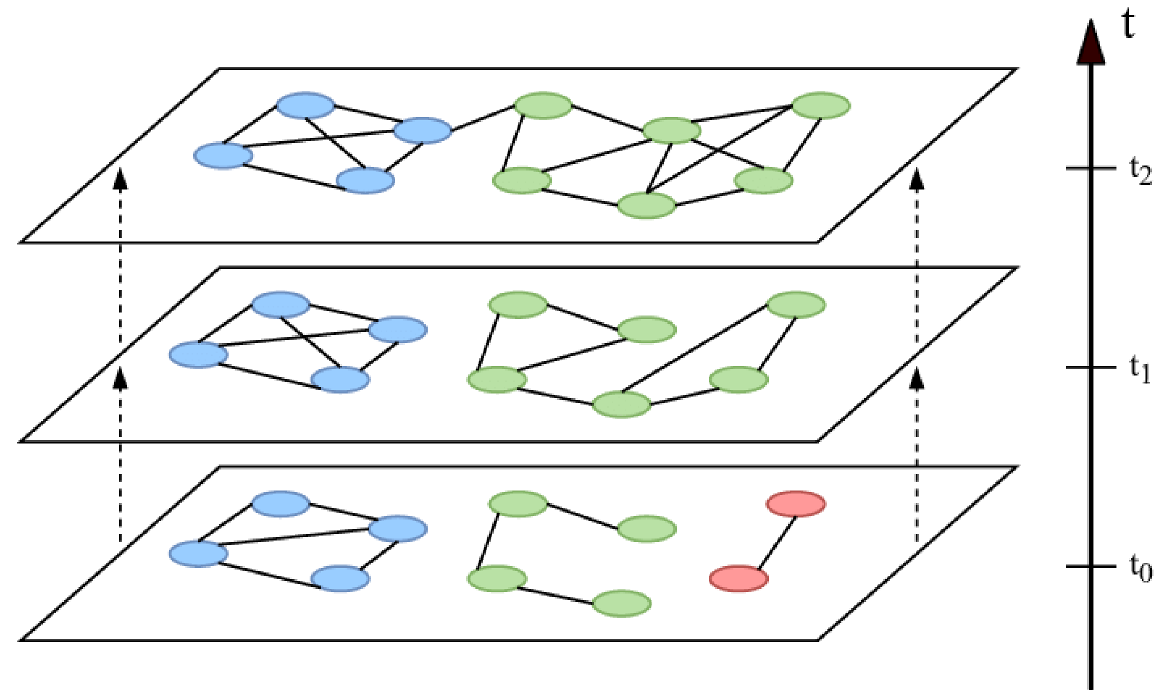
Topics

1. Graphs
   a) Directed Graph
   b) Undirected graphs
2. Applications of Graphs
3. Bounds on the number of edges
4. Degree of a vertex
5. Weighted Graphs
6. Handshaking Lemma
7. Adjacency-matrix representation
8. Adjacency-list representation
9. Graphs in C
10. Using Adjacency Matrix for Path Matrix
11. Warshall's Algorithm
12. Graph Representation

Adjacency Matrix And List Representation

- The most straightforward way to store a graph is in the form of an **adjacency list** or **adjacency matrix.**
- There are multiple ways to store a time-evolving graph while preserving its temporal structure.





Source

Source

Adjacency Matrix And List Representation

- Choosing the right data model depends on the nature of the data, the type of graph (strongly connected vs weakly connected, sparse or dense graphs, etc.), and the targeted data processing and analytical tasks.
- Typically, a sparse (connected) graph has about as many edges as vertices, and a dense graph has nearly the maximum number of edges.

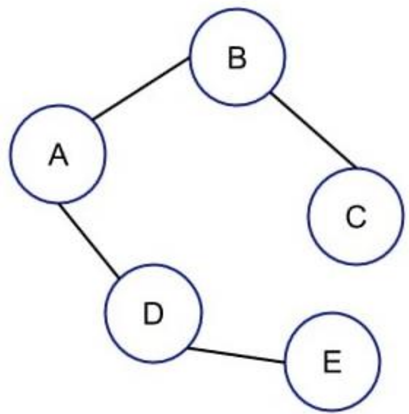


Figure: Sparse Graph

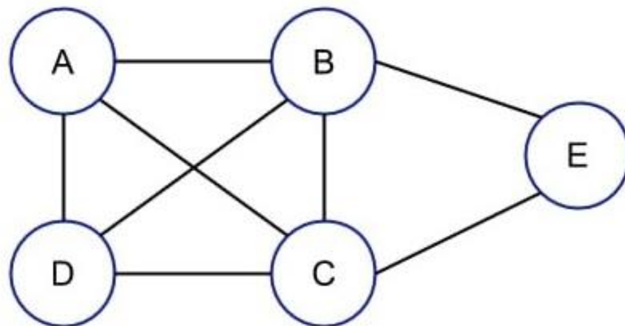
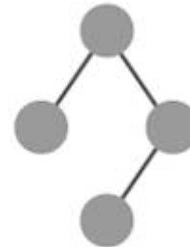
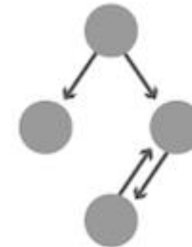


Figure: Dense Graph

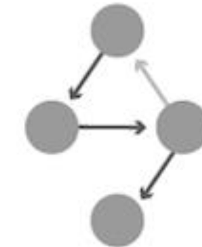
Undirected



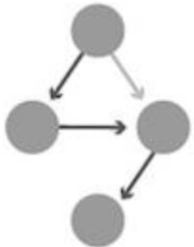
Directed



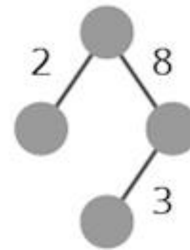
Cyclic



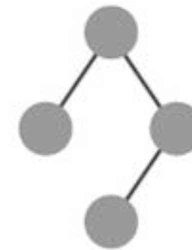
Acyclic



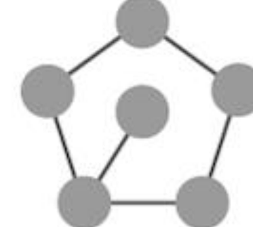
Weighted



Unweighted



Sparse



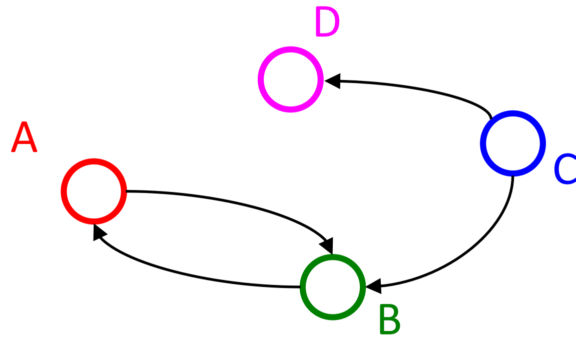
Dense

Adjacency Matrix

- In graph theory and computer science, an **adjacency matrix** is a square matrix used to represent a finite graph.
- The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a-matrix with zeros on its diagonal.

A |V| x |V| matrix of Booleans (or 0 vs. 1, or F vs. T)

- Then $\mathrm{M[u][v]} == \mathrm{true\ (T)}$ means there is an edge from u to v
- Then $\mathrm{M[u][v]} == \mathrm{false\ (F)}$ means there is no edge from u to v



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

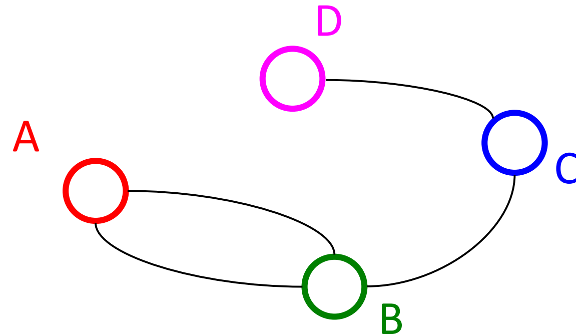
	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	0	0

Note: Number of rows and columns in the matrix = Number of nodes in the graph

Adjacency Matrix Properties

How will the adjacency matrix vary for an **undirected** graph?

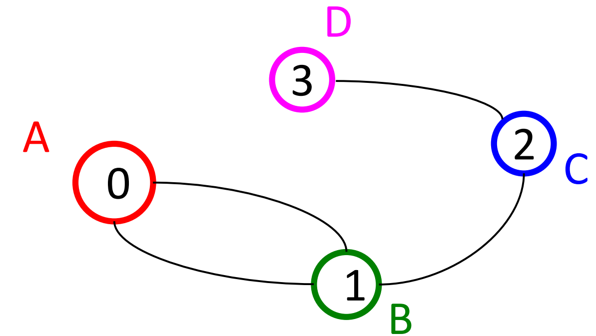
- Will be symmetric about diagonal axis
- Matrix: Could we save space by using only about half the array?



	A	B	C	D
A	F	T	F	F
B	T	F	T	F
C	*F*	*T*	*F*	T
D	*F*	*F*	*T*	*F*

Adjacency Matrix

```c
#include <stdio.h>
#define V 4
void init(int arr[][V]) {
  int i, j; // Initialize the matrix to zero
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}
void addEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1; // Add edges
  arr[j][i] = 1;
}
void printAdjMatrix(int arr[][V]) {
  int i, j; // Print the matrix
  for (i = 0; i < V; i++) {
    printf("%d: ", i);
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  } }
int main() {
  int adjMatrix[V][V];
  init(adjMatrix);
  addEdge(adjMatrix, 0, 1);
  addEdge(adjMatrix, 2, 1);
  addEdge(adjMatrix, 2, 3);
  printAdjMatrix(adjMatrix);
  printf("A=0, B=1, C=2, D=3");
  return 0;
}
```

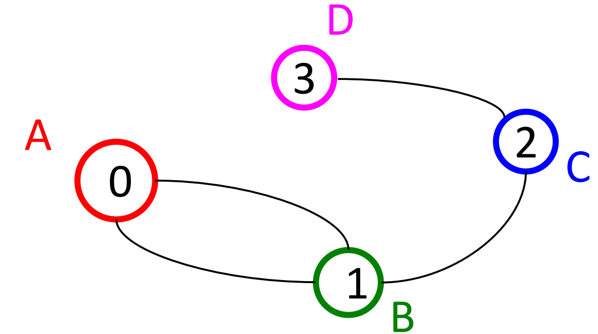


	0	1	2	3
0				
1				
2				
3				

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Adjacency Matrix

```c
#include <stdio.h>
#define V 4
void init(int arr[][V]) {
  int i, j; // Initialize the matrix to zero
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}
void addEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1; // Add edges
  arr[j][i] = 1;
}
void printAdjMatrix(int arr[][V]) {
  int i, j; // Print the matrix
  for (i = 0; i < V; i++) {
    printf("%d: ", i);
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  } }
int main() {
  int adjMatrix[V][V];
  init(adjMatrix);
  addEdge(adjMatrix, 0, 1);
  addEdge(adjMatrix, 2, 1);
  addEdge(adjMatrix, 2, 3);
  printAdjMatrix(adjMatrix);
  printf("A=0, B=1, C=2, D=3");
  return 0;
}
```



0	1	2	3

```
0: 0 1 0 0
1: 1 0 1 0
2: 0 1 0 1
3: 0 0 1 0
A=0, B=1, C=2, D=3
```

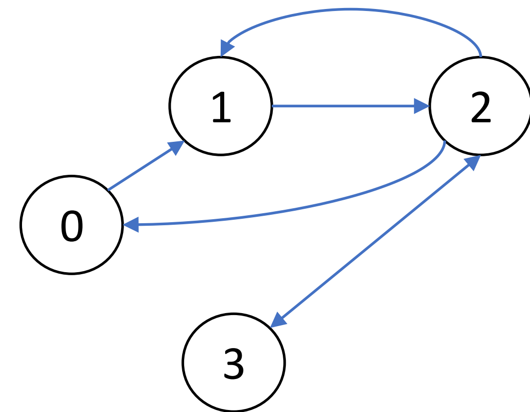
	A	B	C	D
A	F	T	F	F
B	T	F	T	F
C	F	T	F	T
D	F	F	T	F

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

More then 2 edges for a node (Directed Graph)

```
(0 —> 1)
(1 —> 2)
(2 —> 3)        (2 —> 1)        (2 —> 0)
(3 —> 2)
```

```
78    struct Edge edges[] =
79    {
80        {0, 1}, {1, 2}, {2, 0}, {2, 1}, {3, 2}, {2, 3}
81    };
82
83    // calculate the total number of edges
84    int n = sizeof(edges)/sizeof(edges[0]);
85
86    // construct a graph from the given edges
87    struct Graph *graph = createGraph(edges, n);
88
89    // Function to print adjacency list representation of a graph
90    printGraph(graph);
```



Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Advantages and Disadvantages of Adjacency Matrix

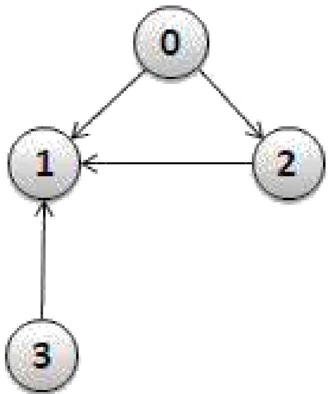
- Advantages:
  - The basic operations like adding an edge, removing an edge, and checking whether there is an edge between nodes are extremely time efficient, constant time operations.
  - If the graph is dense and the number of edges is large, an adjacency matrix should be the first choice.
  - The biggest advantage, however, comes from the use of matrices.
    - The recent advances in hardware enable us to perform even expensive matrix operations on the GPU.
  - By performing operations on the adjacent matrix, we can get important insights into the nature of the graph and the relationship between its vertices.

- Disadvantages:
  - The space requirement of the adjacency matrix requires a lot of memory.
    - Graphs usually don't have too many connections and this is the major reason why adjacency lists (next topic) are the better choice for most tasks.
  - While basic operations are easy, operations like inEdges and outEdges are expensive when using the adjacency matrix representation.
    - i.e. To implement the operations inEdge and outEdge all the entries n, of the corresponding row or column of matrix a[i][j] will have to be scanned

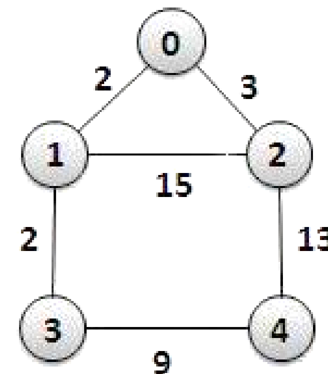
Adjacency Matrix Properties

How can we adapt the representation for weighted graphs?

- Instead of Boolean, store a number in each cell
- Need some value to represent 'not an edge'
  - 0, -1, or some other value based on how you are using the graph
  - Might need to be a separate field if no restrictions on weights



	0	1	2	3
0				
1				
2				
3				

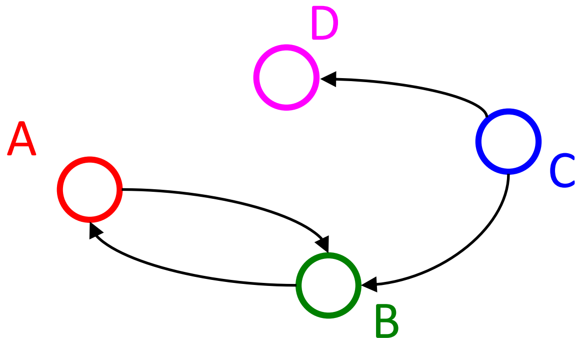


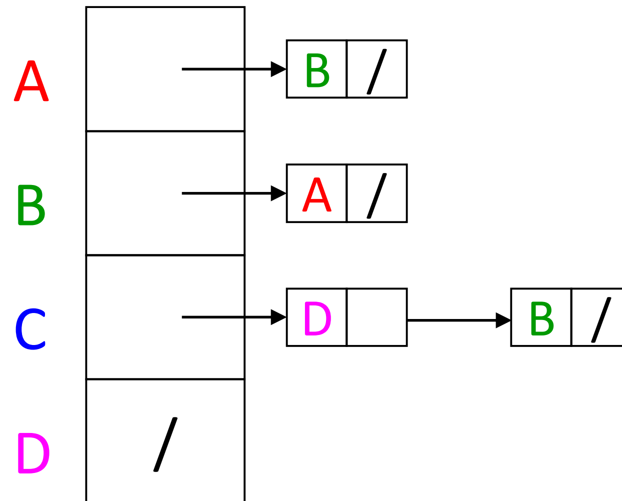
	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

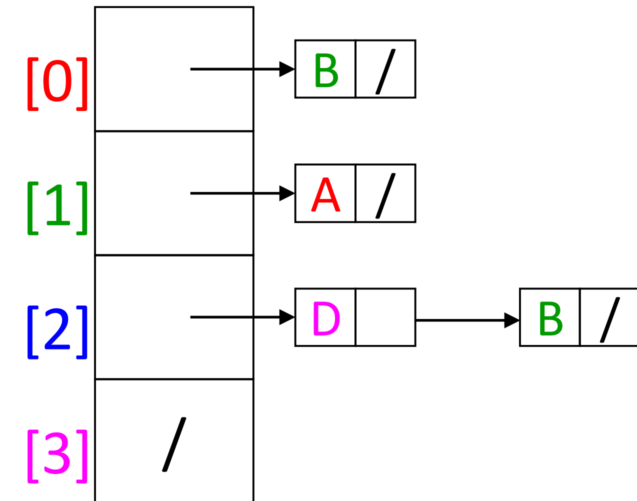
Adjacency List

Assign each node a number from 0 to $|V|$-1

- An array of length $|V|$ in which each entry stores a list of all adjacent vertices (e.g., linked list)
- In Adjacency List, we use an array of a list to represent the graph.
  - The list size is equal to the number of vertex(n).
  - Let's assume the list of size n as Adjlist[n]
  - Adjlist[0] will have all the nodes which are connected to vertex 0.
  - Adjlist[1] will have all the nodes which are connected to vertex 1 and so on.

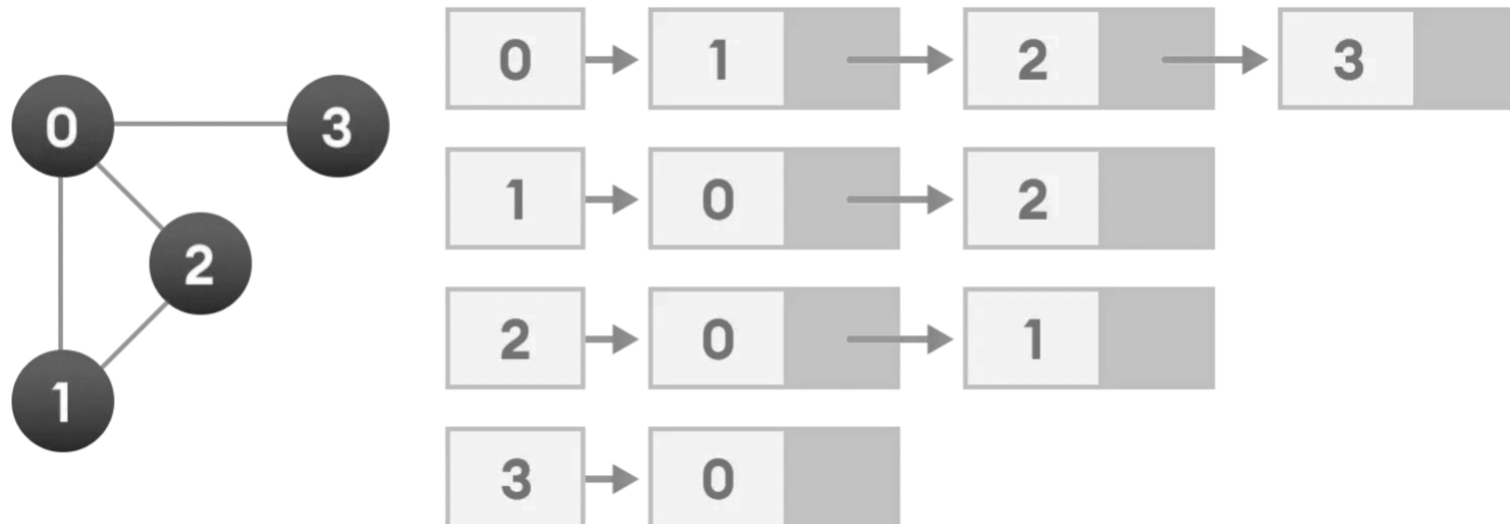


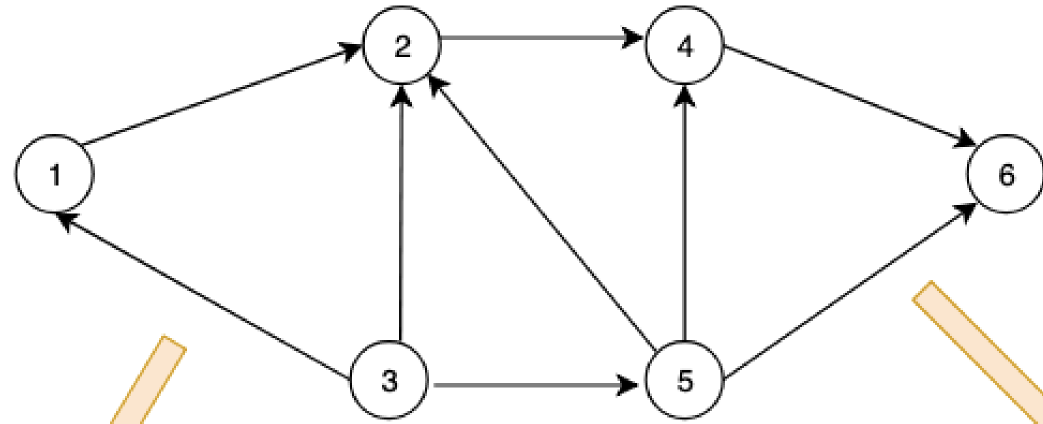




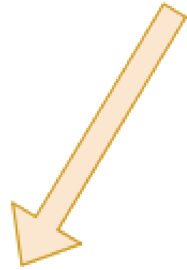
Adjacency List

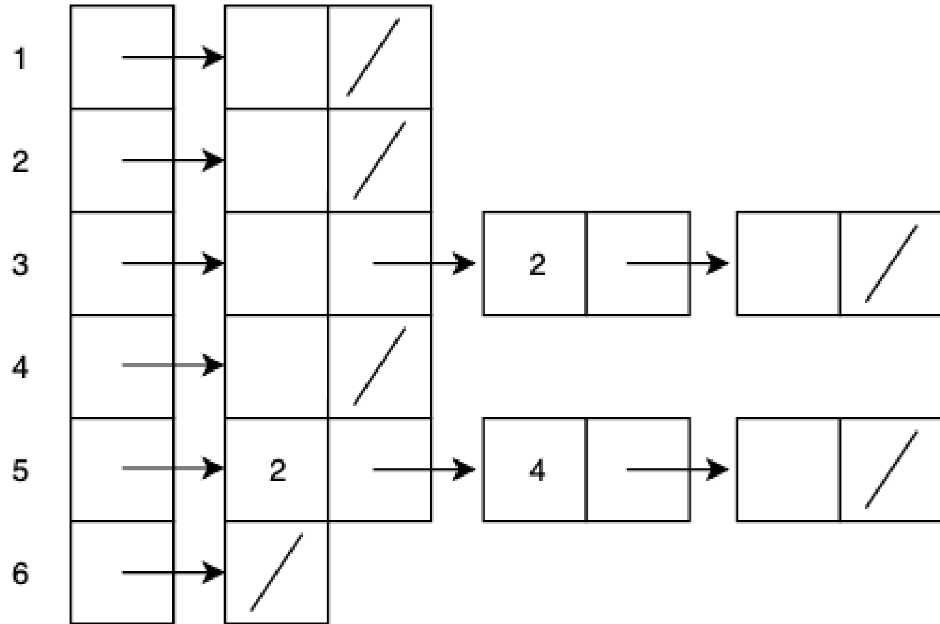
- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.





Adjacency
List



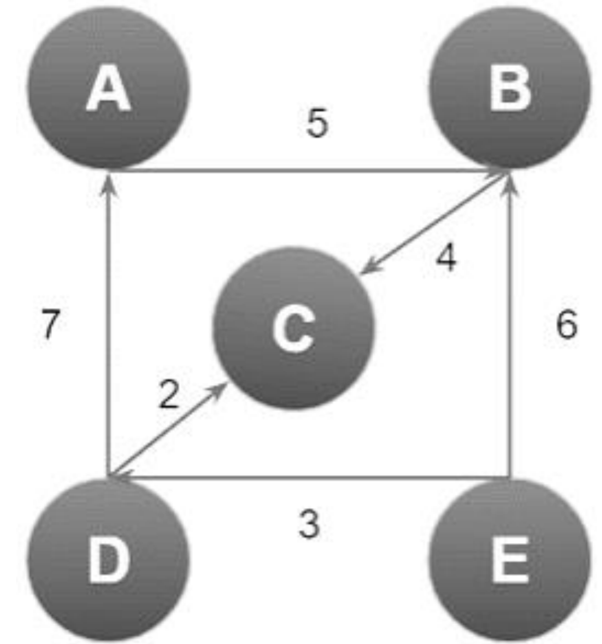


Adjacency
Matrix

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

Adjacency Matrix

- Adjacency Matrix implementation using 2D array:
- **Step 1)** Vertices A has a direct edge with B, and the weight is 5. So, the cell in row A and column B will be filled with 5. The rest of the cells in row A will be filled with zero.
- **Step 2)** Vertices B have a direct edge with C, and the weight is 4. So, the cell in row B and column C will be filled with 4. The remaining cells in row B will be filled with zero as B has no outgoing edge to any other nodes.
- **Step 3)** Vertices C have no direct edges with any other vertices. So, row C will be filled with zeros.

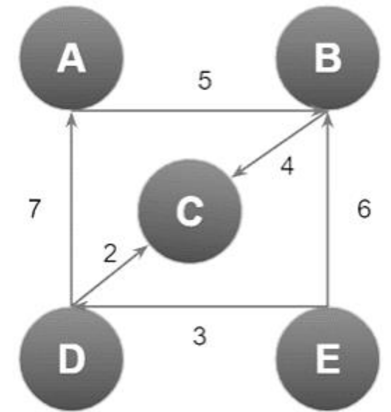


	A(0)	B(1)	C(2)	D(3)	E(4)
A(0)	0	5	0	0	0
B(1)	0	0	4	0	0
C(2)	0	0	0	0	0
D(3)	7	0	2	0	0
E(4)	0	6	0	3	0

The space complexity using the Adjacency matrix will be O(N2), where N means the number of nodes in the Graph

Adjacency Matrix

- Adjacency Matrix implementation using 2D array:
- **Step 1)** Vertices A has a direct edge with B, and the weight is 5. So, the cell in row A and column B will be filled with 5. The rest of the cells in row A will be filled with zero.
- **Step 2)** Vertices B have a direct edge with C, and the weight is 4. So, the cell in row B and column C will be filled with 4. The remaining cells in row B will be filled with zero as B has no outgoing edge to any other nodes.
- **Step 3)** Vertices C have no direct edges with any other vertices. So, row C will be filled with zeros.
- **Step 4)** Vertices D has a directed edge with A and C.
    - Here, the cell in row D and column A will have a value of 7. Cells in row D and column C will have a value of 2.
    - The rest of the cells in row D will be filled with zeros.
- **Step 5)** Vertices E has a directed edge with B and D. The cell in row E and column B will have a valu of 6. Cells in row E and column D will have a value of 3. The rest of the cells in row D will be fille with zeros.



	A(0)	B(1)	C(2)	D(3)	E(4)
A(0)	0	5	0	0	0
B(1)	0	0	4	0	0
C(2)	0	0	0	0	0
D(3)	7	0	2	0	0
E(4)	0	6	0	3	0

	Col1	Col2	Col3	Col4
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]

The space complexity using the Adjacency matrix will be O(N2), where N means the number of nodes in the Graph

Adjacency List

- Adjacency List: An array of linked lists is used. Size of the array is equal to number of vertices.
- Let the array be array[].
  - An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
  - This representation can also be used to represent a weighted graph.
  - The weights of edges can be stored in nodes of linked lists.

```
// Function to print adjacency list representation of a graph
void printGraph(struct Graph* graph)
{
    for (int i = 0; i < N; i++)
    {   // print current vertex and all its neighbors
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("%d -> %d (%d)\t", i, ptr->dest, ptr->weight);
            ptr = ptr->next;
        }
         printf("\n");
    } }
// Weighted Directed graph implementation in C
int main(void)
{    // input array containing edges of the graph (as per the above diagram)
    // (x, y, w) tuple represents an edge from x to y having weight `w`
    struct Edge edges[] =
    {
        {0, 1, 6}, {1, 2, 7}, {2, 0, 5}, {2, 1, 4}, {3, 2, 10}, {4, 5, 1}, {5, 4, 3}
    };
```

```
0 —> 1 (6)
1 —> 2 (7)
2 —> 1 (4)        2 —> 0 (5)
3 —> 2 (10)
4 —> 5 (1)
5 —> 4 (3)
```

Note: This code will not be part of quiz or exam. It is only for implementation and understanding

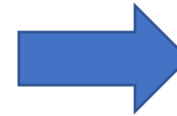
Source

Adjacency List

- Adjacency List: An array of linked lists is used. Size of the array is equal to number of vertices.
- Let the array be array[].
  - An entry array[i] represents the linked list of vertices adjacent to the ith vertex.
  - This representation can also be used to represent a weighted graph.
  - The weights of edges can be stored in nodes of linked lists.

```
0 -> 1 (6)
1 -> 2 (7)
2 -> 1 (4)      2 -> 0 (5)
3 -> 2 (10)
4 -> 5 (1)
5 -> 4 (3)
```

	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						



Source

Pros and Cons of Adjacency List

- Pros
  - An adjacency list is efficient in terms of storage because we only need to store the values for the edges.
    - For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.
  - It also helps to find all the vertices adjacent to a vertex easily.
- Cons
  - Finding the adjacent list is not quicker than the adjacency matrix because all the connected nodes must be first explored to find them.

Adjacency List

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  struct node {
4    int vertex;
5    struct node* next;
6  };
7  struct node* createNode(int);
8  struct Graph {
9    int numVertices;
10   struct node** adjLists;
11 };
12 struct node* createNode(int v) { // Create a node
13   struct node* newNode = malloc(sizeof(struct node));
14   newNode->vertex = v;
15   newNode->next = NULL;
16   return newNode;
17 }
18 struct Graph* createAGraph(int vertices) { // Create a graph
19   struct Graph* graph = malloc(sizeof(struct Graph));
20   graph->numVertices = vertices;
21   graph->adjLists = malloc(vertices * sizeof(struct node*));
22   int i;
23   for (i = 0; i < vertices; i++)
24     graph->adjLists[i] = NULL;
25   return graph;
26 }
```

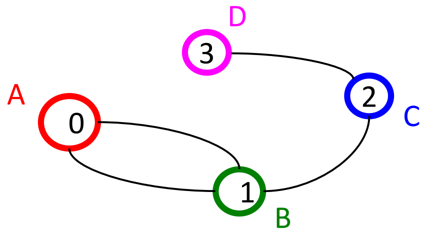
```
27 void addEdge(struct Graph* graph, int s, int d) {
28   //Add edge - Add edge from s to d
29   struct node* newNode = createNode(d);
30   newNode->next = graph->adjLists[s];
31   graph->adjLists[s] = newNode;
32   // Add edge from d to s
33   newNode = createNode(s);
34   newNode->next = graph->adjLists[d];
35   graph->adjLists[d] = newNode;
36 }
37 void printGraph(struct Graph* graph) {
38   int v; // Print the graph
39   for (v = 0; v < graph->numVertices; v++) {
40     struct node* temp = graph->adjLists[v];
41     printf("\n Vertex %d\n: ", v);
42     while (temp) {
43       printf("%d -> ", temp->vertex);
44       temp = temp->next;
45     }
46     printf("\n");
47   } }
48 int main() {
49   struct Graph* graph = createAGraph(4);
50   addEdge(graph, 0, 1);
51   addEdge(graph, 2, 1);
52   addEdge(graph, 2, 3);
53   printGraph(graph);
54   return 0;
55 }
```

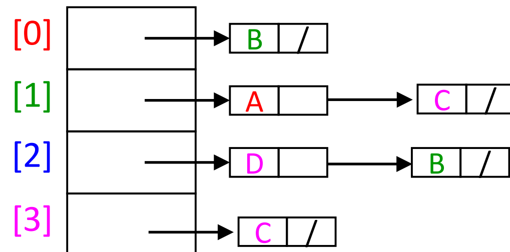
```
Vertex 0
: 1 ->

Vertex 1
: 2 -> 0 ->

Vertex 2
: 3 -> 1 ->

Vertex 3
: 2 ->
```

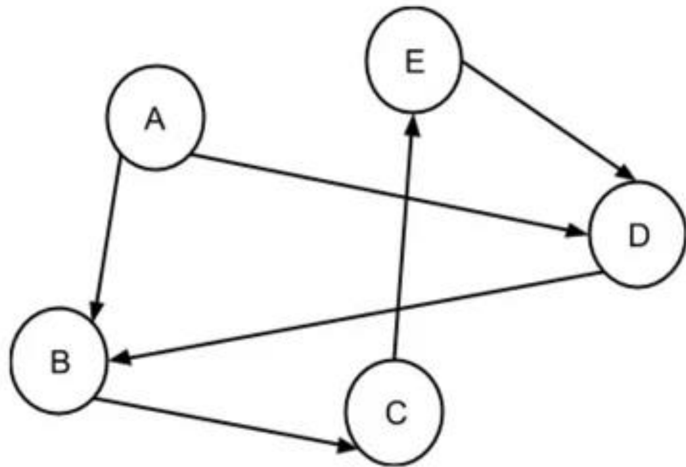




Note: This code will not be part of quiz or exam. It is only for implementation and understanding

Path Matrix in Graph Theory

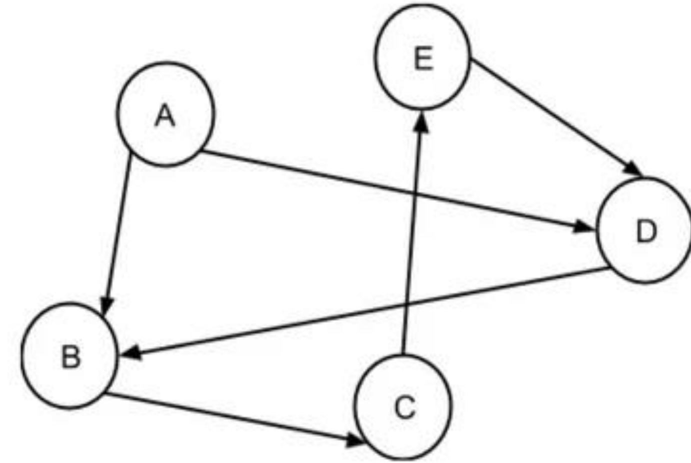
- A path matrix in a data structure is a matrix representing a graph, where each value of a row (A) and a column (B) project whether there is a path from node A to node B.
- The path may be direct or indirect.
- It may have a single edge or multiple edges.



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

M

Path Matrix in Graph Theory



- If we consider the pair of nodes B and E.

- From the adjacency matrix, we find that there is no direct path between B and E.

  - But there exists a path from B and E through C.
  - Such a path is known as a path of length 2.

- Similarly, a path of length 3 will have 2 intermediate nodes.

- In general, a path of length n will have (n – 1) intermediate nodes.
- To obtain a path matrix of length 2, the adjacency matrix is multiplied by itself.

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

M

Path Matrix in Graph Theory

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

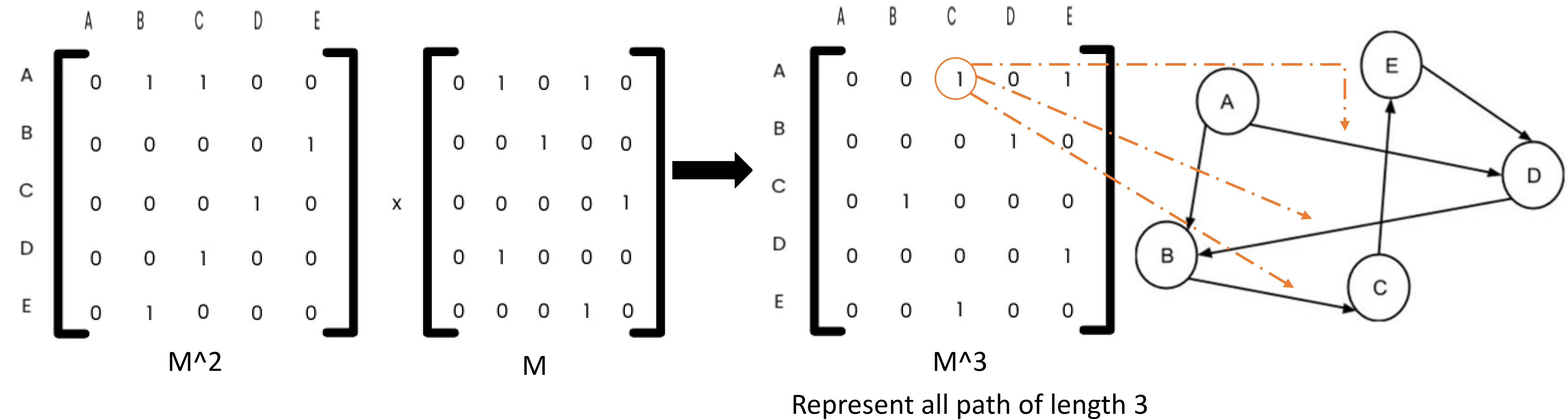
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \times \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \Rightarrow \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

M M M^2

Represent all path of length 2

- From the resultant matrix (M^2), we find that there exists a path of length 2 between A to B, E to B, A to C, D to C, C to D, and B to E.
- Again multiplying the path matrix of length 2 with the adjacency matrix gives a path matrix of length 3.

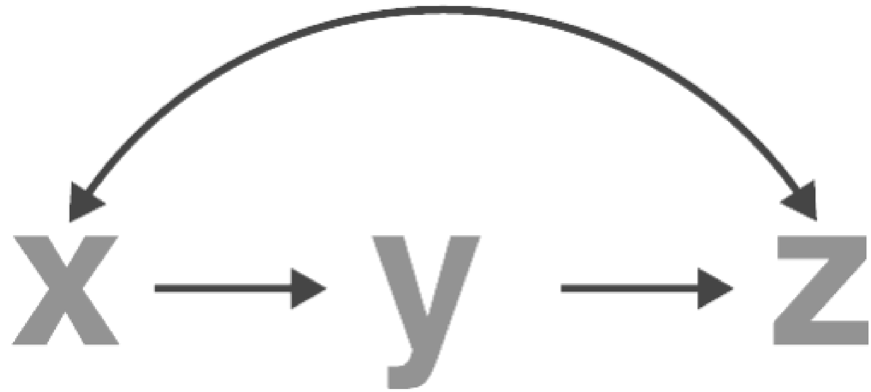
Path Matrix in Graph Theory



- From the resultant matrix (M^3), we find that there exists a path of length 3 between C to B, A to C, E to C, B to D, A to E, and D to E.
- In general, to generate the matrix of the path of length n, take the matrix of the path of length (n – 1), and multiply it with the matrix of a path of length 1.

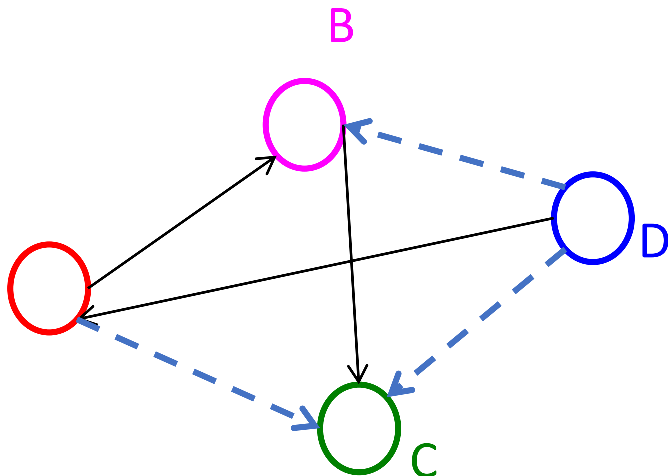
Warshall's Algorithm

- **Transitive Relation:** A relation R is transitive on a set S if for all x, y, z ∈ S, if:
  - (x, y) ∈ R and (y, z) ∈ R, then (x, z) ∈ R.
  - In a graph of a reflexive relation, every node will have an arc back to itself.



Warshall's Algorithm

- **Transitive Closure:** Given a relation r on a set A, the transitive closure of r is the smallest transitive relation that contain r as a subset (notation: r*)
- The transitive closure of a graph is a compact representation of all the reachability information in the graph.
  - There are different methods to compute the transitive closure of a graph, including the Floyd-Warshall algorithm, the dynamic programming approach, and the matrix multiplication method



True Relation: (A, B), (B, C), (D, A) - graph is not transitive

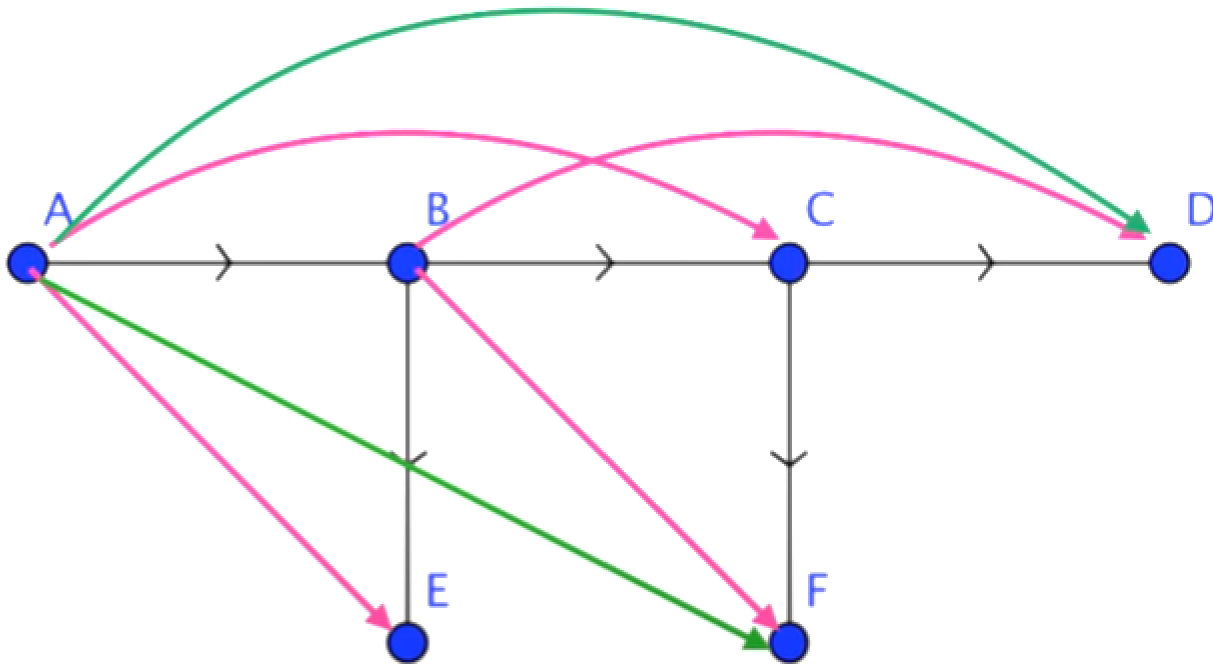
- Algorithm (Social Media/online shopping) Suggest: (A, C), (D, B)
- New edges (A,C) and (D,B)
- In a transitive relation, direct relation should exist (not 2nd level or 2 edge traversal)

By adding (D, C) we have created new relation in the graph and made it transitive

{(A, B), (B, C), (D, A), (A,C), (D,B), (D, C)} = The original pair of edges ∈ New set

Warshall's Algorithm

- Practically we need an algorithm to help us add edges on a graph to make it transitive (as in real world we might be working with hundreds of nodes and edges)

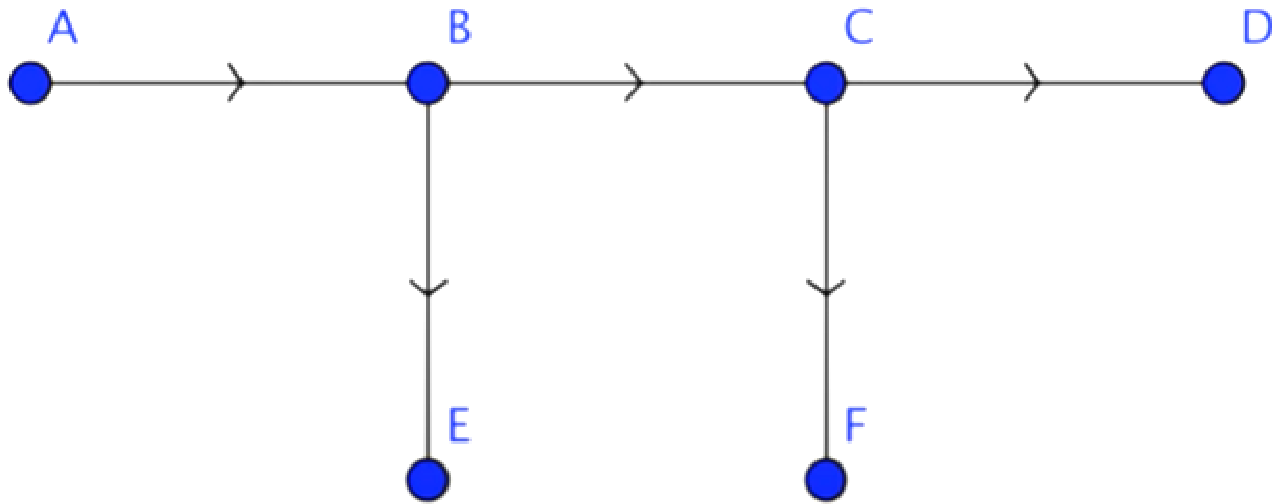


Now we have new relations due to the pink edges we added
But now we have another concern!

Based on number of nodes (in this graph) we need to loop through the graph 6 times to achieve transitive closure

Warshall's Algorithm

- Practically we need an algorithm to help us add edges on a graph to make it transitive (as in real world we might be working with hundreds of nodes and edges)
- As computer need matrix based representation to work

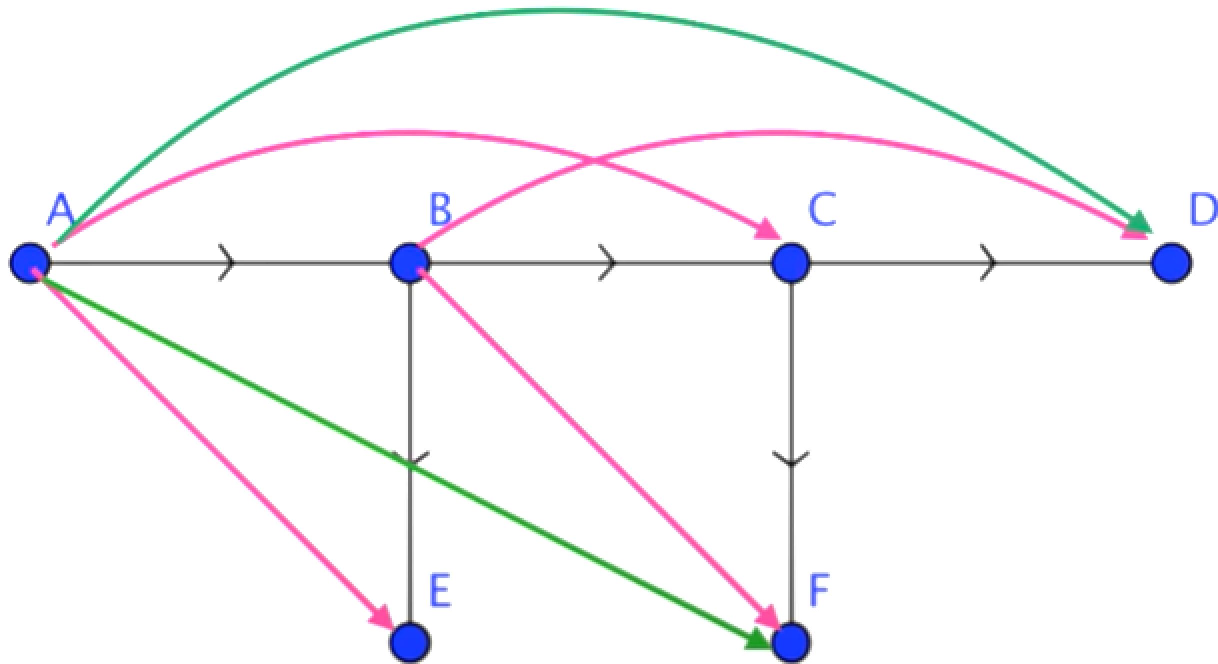


Original edges are preserved by the algorithm

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Warshall's Algorithm

- Practically we need an algorithm to help us add edges on a graph to make it transitive (as in real world we might be working with hundreds of nodes and edges)
- As computer need matrix based representation to work

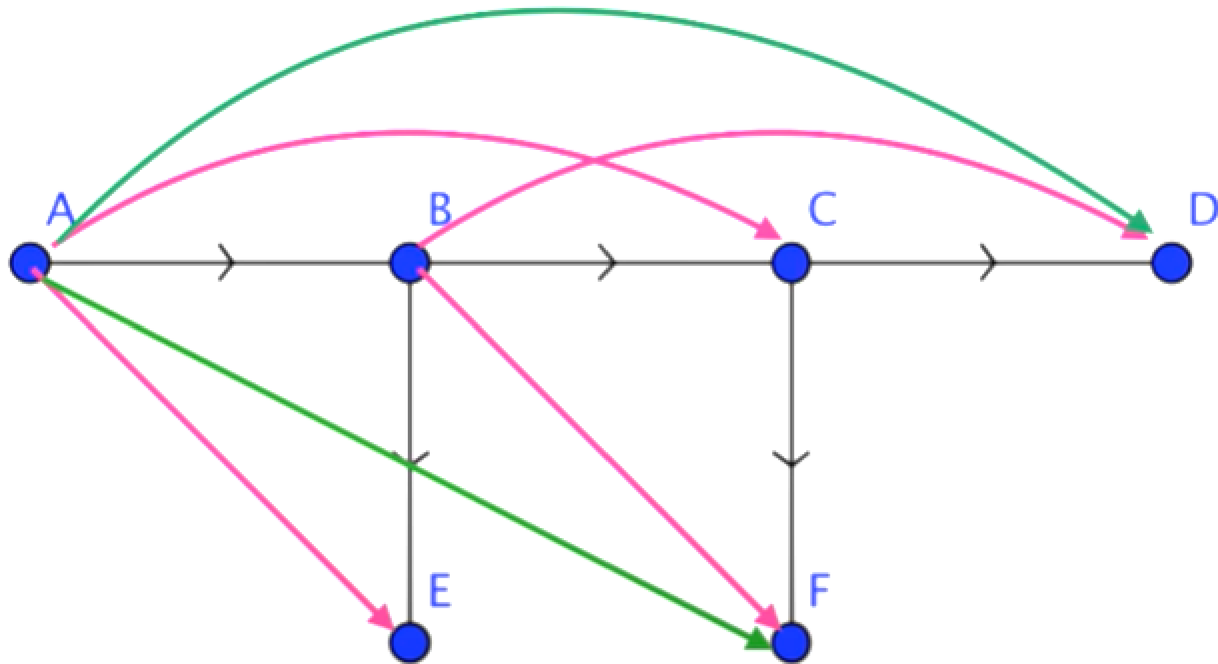


Update the edges in the Adj-Matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Warshall's Algorithm

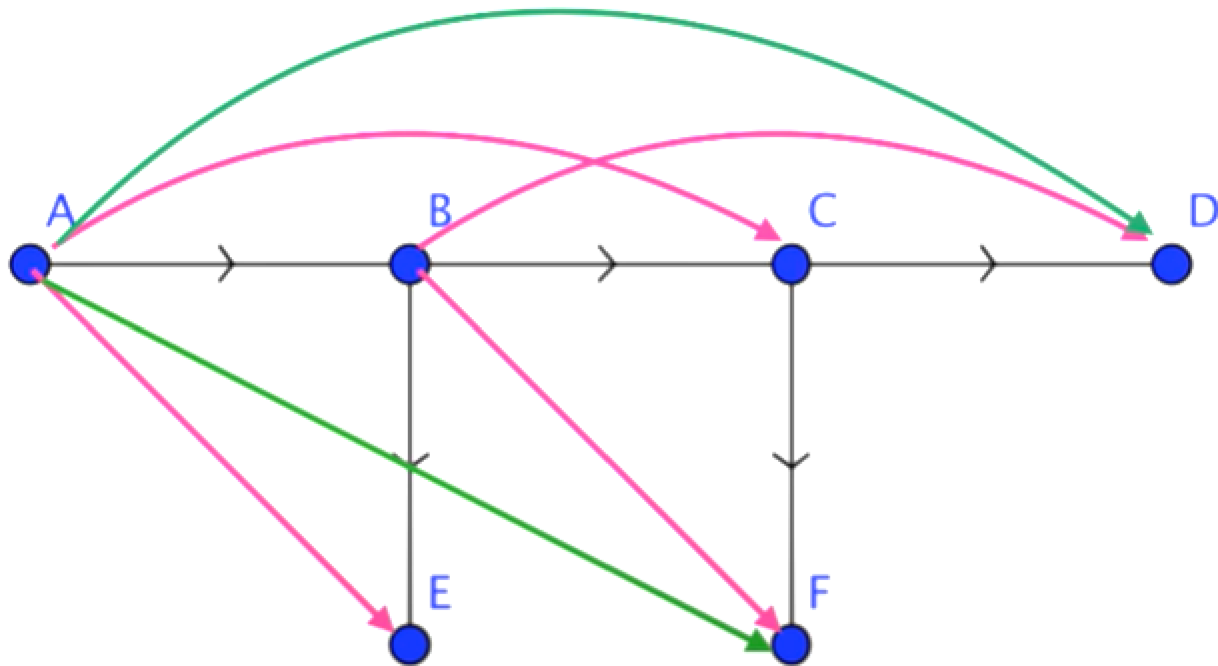
- Practically we need an algorithm to help us add edges on a graph to make it transitive (as in real world we might be working with hundreds of nodes and edges)
- As computer need matrix based representation to work



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Warshall's Algorithm

- Practically we need an algorithm to help us add edges on a graph to make it transitive (as in real world we might be working with hundreds of nodes and edges)
- As computer need matrix based representation to work

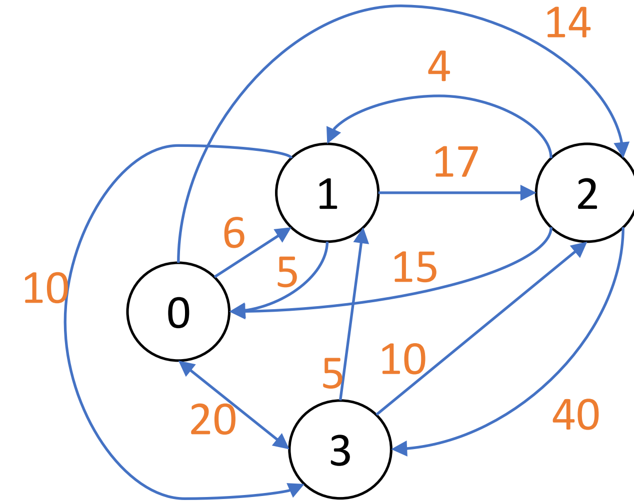


Second Round

	A	B	C	D	E	F
A	0	1	1	1	1	1
B	0	0	1	1	1	1
C	0	0	0	1	0	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Warshall's Algorithm
Application

```
Enter the number of vertices: 4
Enter the edges:
[0][0]: 0
[0][1]: 6
[0][2]: 14
[0][3]: 20
[1][0]: 5
[1][1]: 0
[1][2]: 17
[1][3]: 10
[2][0]: 15
[2][1]: 4
[2][2]: 0
[2][3]: 40
[3][0]: 20
[3][1]: 5
[3][2]: 10
[3][3]: 0
```




```
The original graph is:
0 6 14 20
5 0 17 10
15 4 0 40
20 5 10 0
```

	0	1	2	3
0	*0*	*6*	*14*	*20*
1	*5*	*0*	*17*	*10*
2	*15*	*4*	*0*	*40*
3	*20*	*5*	*10*	*0*

Warshall's Algorithm (Example)

Source

Warshall's Algorithm (Example)

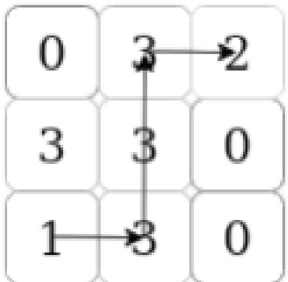
- **0** = Blank Wall.
- **1** = Source.
- **2** = Destination.
- **3** = Blank cell.

- M[3][3] =

{{ 0, 3, 2 },

{ 3, 3, 0 },

{ 1, 3, 0 }};



1. Traverse the matrix and find the starting index of the matrix.
2. Create a recursive function that takes the index and visited matrix.
3. Mark the current cell and check if the current cell is a destination or not. If the current cell is the destination, return true.
4. Call the recursion function for all adjacent empty and unvisited cells.
5. If any of the recursive functions returns true then unmark the cell and return true else unmark the cell and return false.